Spring 2015

Dynamic textures

Illia Ziamtsov
Purdue University

Graduate School Form 30
Updated 1/15/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Illia Ziamtsov

Entitled
DYNAMIC TEXTURES

For the degree of Master of Science

Is approved by the final examining committee:

Bedrich Benes
Chair

James Mohler

Yingjie Chen

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Bedrich Benes

Approved by: Patrich E Connolly 4/27/2015
Head of the Departmental Graduate Program Date

DYNAMIC TEXTURES

A Thesis

Submitted to the Faculty

of

Purdue University

by

Illia Ziamtsov

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2015

Purdue University

West Lafayette, Indiana

TABLE OF CONTENTS

Page

LIST OF FIGURES

ABSTRACT

Ziamtsov, Illia M.S., Purdue University, May 2015. Dynamic textures. Major Professor: Bedrich Benes.

In this research study we introduce a new way to create textures by using (Reynolds, 1987) model. The study builds and extends upon principles outlined by (Reynolds, 1987). The study defines a class of textures that can be generated with boids' behavior. Boids are tested with a combination of vector fields in 2D. The combination produces interesting color and image effects. Movement of boids and generation of textures on a 3D surface are explored as well. A novel way for boids to move on a surface of a 3D model is presented.

CHAPTER 1. INTRODUCTION

Many tools exist today that deal with texture creation and texture generation in the field of 3D graphics. Textures play an important role in almost all branches of computer graphics, one of the most important being they bring more realism and believability to the object on which they are used. Textures today can be created in many different ways for a variety of different purposes. The majority of texture creation can be associated with either a completely manual process or a creation that has full or some automation in it. From the very beginning researchers have attempted to automate the process of a texture creation. The motivation for automating textures is the amount of time it takes an artist to draw the textures by hand where he or she has to paint every single stroke. The textures that have very complex components will usually take a long amount of time to produce. Also there are textures that are very hard or even impossible to produce automatically. On the other hand, there are automatically created textures that are impossible to do manually. There is a gap between what can be done automatically and what cannot. It would be significant to extend the reach of the automatic methods to be able to automate things that would have to be done manually and vice versa.

In this thesis, we propose a method that attempts to extend the current automatic methods by utilizing a phenomena known as emerging behavior. The emerging behavior is a common phenomena in nature. It can be found by observing birds, fish, and ants to mention a few. Previous work from (Beneš, 2006), (Kopf et al., 2007), and (Reynolds, 1987) was already conducted on a variety of subjects spanning from the simulation of birds to the simulation of traffic jams. The emerging behavior will be responsible for creating complex textures based on a particle system that would be very hard and time consuming to create manually.

1.1 Scope

The research work presented in this thesis covers the exploration of the phenomena called emerging behavior as applied to texture generation. The emerging behavior is achieved through utilization of the particle system. The research explores a variety of different constraints that govern the movement of the particles in 2D space as well as on the surface of the 3D model. A part of the project is to find useful and appropriate constraints or rules to achieve meaningful patterns. The second part of the project is to figure out an efficient technique for particle communication and interaction. The third and the final part of the research is to evaluate and compare different rules and visual qualities against the efficiency associated with them. Algorithmic efficiency of the application will be tested.

1.2 Significance

This project will extend the knowledge on the subject of automatic texture generation by creating a framework that will automatically generate textures based on a particle system. The resulting method will not only be faster as opposed to a manual method but also allow a high degree of control. The framework will be able to generate complex patterns, which would be hard to do using traditional methods. The tool might save time and effort for an artist using it and be a good addition to more traditional methods.

1.3 Hypotheses

The hypotheses for this study are the following:

H_1: if a boid's behavior is used then procedural texture can be generated.

H_1: if a boid's behavior is used in 3D a segregated type texture can be created.

1.4 Research Question

Can a class of textures be generated with a boid behavior model?

Can a boid's texture generation be extended and mapped a to mesh in 3D to create a segregated texture?

1.5 Assumptions

The assumptions for this study include:

- The input mesh has a valid format that contains connectivity information.
- The user has some familiarity working with 3D graphics.
- The computer hardware works according to the specified settings.

1.6 Limitations

The limitations for this study include:

- The system takes as an input a closed triangulated mesh.
- The mesh has one surface.
- The maximum number of elements that can be generated is restricted by the amount of memory.

1.7 Delimitations

The delimitations for this study include:

- The system uses a distance function to compute the distances between the particles. This study will not focus on distance functions.
- The study will not focus on creating textures for dense meshes.

- This study will not attempt to explain the emergence patterns.

1.8 Definitions

3D Texture – also sometimes called a volume texture is a series of 2D texture put together in a deck of cards manner where each 2D texture is a card or a slice from the deck. 3D textures are used for a volumetric rendering.

Billboard – a technique in computer graphics to create a detail in a 3D scene that is represented by an image that is always orthogonal to a viewing camera.

Boid – stands for bird like object.

Boid layer – a feature that is segregated from a texture and simulated with a boid simulation.

Computer simulation – a computer program which attempts to reproduce some behavior from a real world.

Decals – a pattern or design that will be transferred on a surface of a 3D model in this case.

Diffusion-limited aggregation – a process in which particles are moving according to a Brownian motion and cluster together when they collide with other particles to form aggregates.

Emerging behavior – comes from complex systems that often behave in unexpected ways that are not easily predictable from the behavior of their components (Hillis, 1988).

Geodesic distance – a distance that considers a curvature of a surface.

OBJ format – a format that was adopted in computer graphics to describe geometry. It includes vertexes, vertex normals, texture coordinates, and vertex connectivity information or faces.

Particle system – a system that models an object as a cloud of primitive particles that define its volume. Over a period of time, particles are generated into the system, move and change form within the system, and die from the system. The resulting model is able to represent motion, changes of form, and dynamics that are not possible with classical surface-based representations (Reeves, 1983).

Procedural texture generation – a texture generation technique that is done by using a computer algorithm as apposed to a human.

Texture – in computer graphics texture represents a surface of an object (3D model). A texture not only can represent color and brightness but also it can represent three-dimensional features such as reflection and transparency. The textures are applied to 3D models by texture mapping.

Texture mapping – a process of applying a texture to a 3D model by wrapping it around a surface of a model.

Wireframe mode – a mode in which connectivity between all vertexes in the mesh are clearly marked, so one can see polygon structure of the mesh.

1.9 Summary

This chapter provided the scope, significance, research question, assumptions, limitations, delimitations, definitions, and other background information for the research project. All these components narrowed the focus, revealed the purpose, and allowed for the reevaluation of the direction chosen for this research study. The components of this section will help to keep the study within defined boundaries. Definitions provided here will be referenced throughout the study.

CHAPTER 2. PREVIOUS WORK

This chapter provides a review of the relevant literature on the topics of particle systems and textures. The review builds a case for the research project.

2.1 Introduction

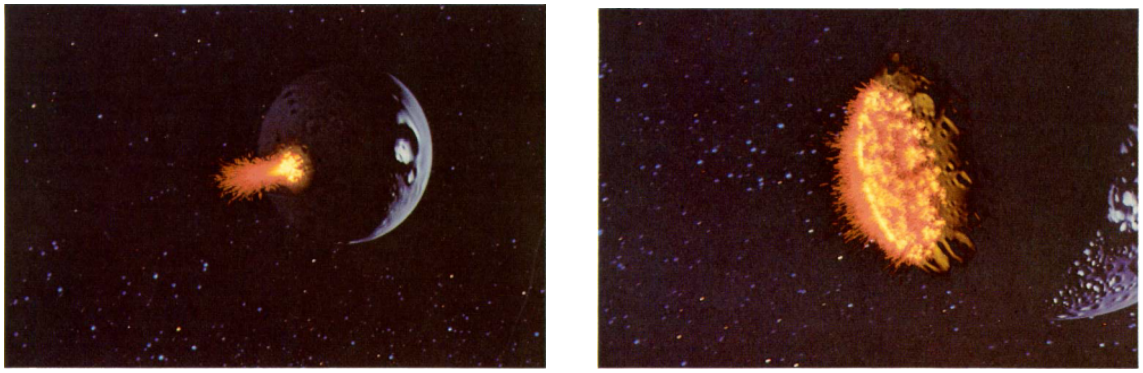
Throughout the history of computer graphics, particles in all their shapes and forms have had a lot of attention from researchers. Today particles remain a highly researched field in a variety of different contexts. A number of sub fields in computer graphics employ particles. Some of the classic uses of particles include fluids, fire, clouds, snow, and rain simulation. In order for particles to be useful, there must be many of them in a scene; collectively this is called a particle system.

Textures have been around in computer graphics for a long time as well. They bring more realism to the overall scene as opposed to the monotonous, perfect-looking 3D objects of early computer graphics. Textures have been researched from a variety of different angles. A combination and synthesis of both textures and particles brings an interesting new visual quality and control to a user. The next two sections provide a brief overview of the work that was done on both subjects.

2.2 Particle System

The term particle system was defined by (Reeves, 1983). The author stated that a particle system is the number of particles that model an object by defining its volume. Over the course of time, particles can be generated, moved around, changed in form, or die within the system. The resulting outcome is able to

simulate dynamics, change of form, and various motions that would be difficult to achieve using surface based methods such as polygons or patches. An example of a particle simulation is shown in Figure 2.1.



(a) Initial explosion (b) Expanding wall of fire

Figure 2.1. Examples of fuzzy objects created with particles (Reeves, 1983).

Particle systems remain important for three main reasons as stated by (Reeves & Blau, 1985). The first reason is that particles are a lot simpler than other computer graphics primitives. As a result a lot more can be drawn with particles within given computational resources. The second reason is particles might be generated not only by procedural methods but also by stochastic methods. Lastly, particles can be used to simulate things that change over time. These reasons give a substantial advantage to particle systems as opposed to surface methods mentioned above; however, there are also limitations of particle systems.

(Reeves & Blau, 1985) wrote one of the first papers that tried to take a step towards justifying a more generic use for a particle system. In this paper, the authors proposed a novel approach to approximate and probabilistic algorithms for rendering and shading particles. The authors also realized the limitation of a particle system; in particular they recognized particles produce more irregular three-dimensional detail as opposed to surface methods. The exact shading on this

irregular detail becomes impossible, which is why the authors tried to approximate the detail. The authors looked at the particle as the smallest building block for the scene. Next, they generated a scene where all the objects were constructed solely by particles. The objects included things such as grass, trees, and terrain. In the end, the authors tried to compare the visual quality and efficiency of the images created with particles and without particles. The authors discovered that it was very hard to make the comparison accurately; however the time to generate the particles was not tremendously computationally expensive. This particular paper did not cover textures however it is an example that particle systems can be very powerful.

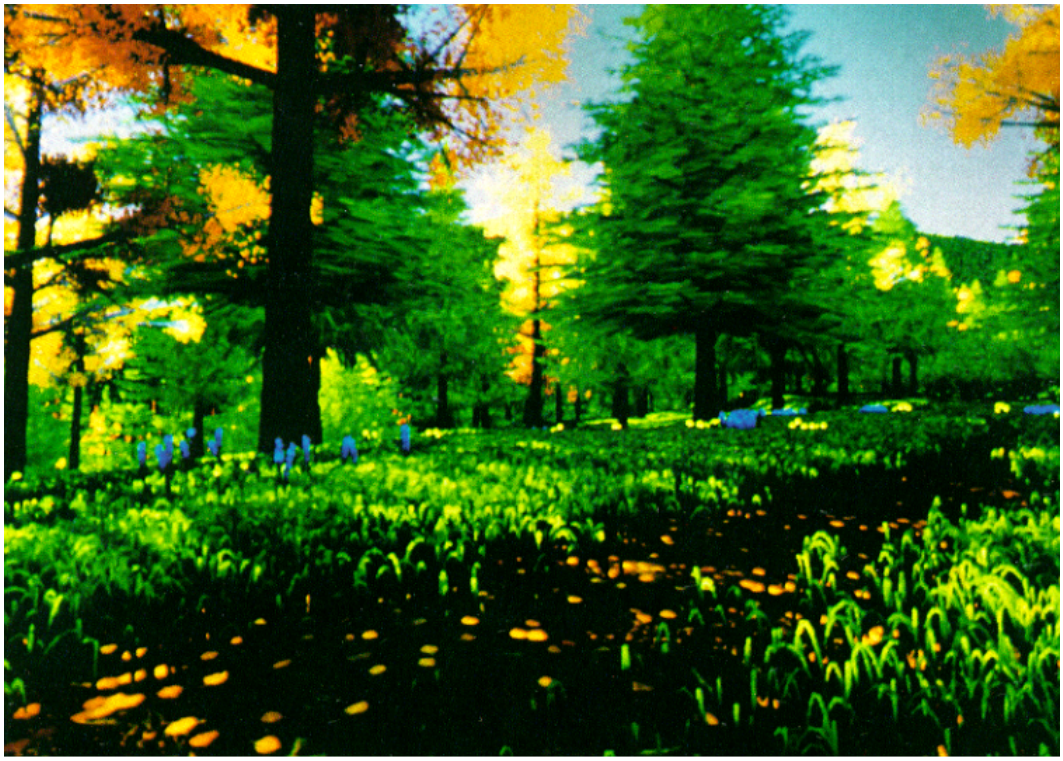


Figure 2.2. Forest Scene from *The Adventures of Andre and Wally B.* An example of a scene generated solely by particles (Reeves & Blau, 1985).

Another paper on particles published by (Bourke, 2006) demonstrates another interesting method involving particles. The method extends something known as diffusion-limited aggregation. Diffusion-limited aggregation was first

introduced by (Witten & Sander, 1983). The rules of diffusion-limited aggregation are very simple. First a particle is introduced into the space and it randomly moves around until it hits an existing structure, which at the very beginning is just another point that is stationary. When the stationary point is hit, it becomes a part of the structure and the same process repeats many times. As the branches grow out, it is easy to predict that the growth will occur at the very tips of the branches because as they grow it will be more difficult for other particles to get inside, closer to the center. Diffusion-limited aggregation has been studied as a 2D model for many years in subjects such as networks of rives, plant growth, corals, lighting, and electro-decomposition.

The main contribution of this paper is the application of diffusion-limited aggregation, or DLA, into 3D space. The method also implements a way to constrain a DLA by a surface or having it inside a vessel. In one of the examples, the author places a DLA into a cylinder-like container with one open end on top so the branching is constrained by the walls of the container and grows only into the direction of open space. The behavior that can be observed by looking at growth of the DLA resembles a fractal behavior.

Particles can simulate very complex behaviors that otherwise would be very hard to produce. In the paper by (Reynolds, 1987), the author attempted to use a particle system to simulate the complex behavior of birds and fish. The paper is based on the fact that both birds and fish are creatures that fly or swim in flocks and flocks behave as one unit, which helps them to survive. In his system, each bird or fish is represented as a particle that moves in space. The author calls these objects boids. The space in which the boids interact can be 2D or 3D.

The most important contribution of the paper is the way the boids interact and move in the system, which very closely resembles real life behavior. The movement of boids is defined by three simple rules: separation, cohesion, and alignment. Each of these rules is calculated based on the neighboring boid's position and velocity for every bird in the flock. It is important to note that each boid has

very limited visibility and only the boids that are inside of the circle of vision are considered for calculating the rules for the boid.

The author defines the rule of separation as the inverse of the sum of vectors from the boid to each of its neighbors. It computes where most of the boids are and creates a force that pushes the boid in the opposite direction. The separation makes sure that the boids do not crash into each other. The rule of cohesion does exactly the opposite and tries to pull the boids together to maintain the flock formation. It calculates the average position from the current boids' positions in the circle of vision and then it forces the boid into the direction of that average position. The rule of alignment is responsible for making sure that the flock has a uniform movement in some direction. Instead of averaging the positions of boids like it is performed in cohesion, the velocities of the boids are averaged. This gives the flock the unit movement mentioned above. In the end, all the resulting vectors from the rules are summed into one force and applied to each boid. This technique produces a realistic animation of boids.

The behavior that evolves from these rules is also called emerging behavior. Emerging behavior is a collective behavior of objects that is hard to predict by knowing the behavior of the individual object. The object usually has a very limited vision of the world around it but because the vision collectively propagates to other neighbors, the higher order of behavior occurs. Examples of this phenomenon in nature would be ant and bee colonies.

A number of attempts to extend the behavior of boids was done as well as applying the behavior to other objects besides fish and birds. The paper written by (Beneš, 2006) attempts to expand the rules. The authors came up with a novel method of adding an additional complementary rule that resembles the behavior of some birds that are on the edge of a flock to suddenly shoot off away from the flock. In addition they also added a concept of leadership in the flock. The leadership is defined by a boid's position and eccentricity. The boids that are on the edges of the flock have a higher probability of flying away than the ones on the inside of a flock.

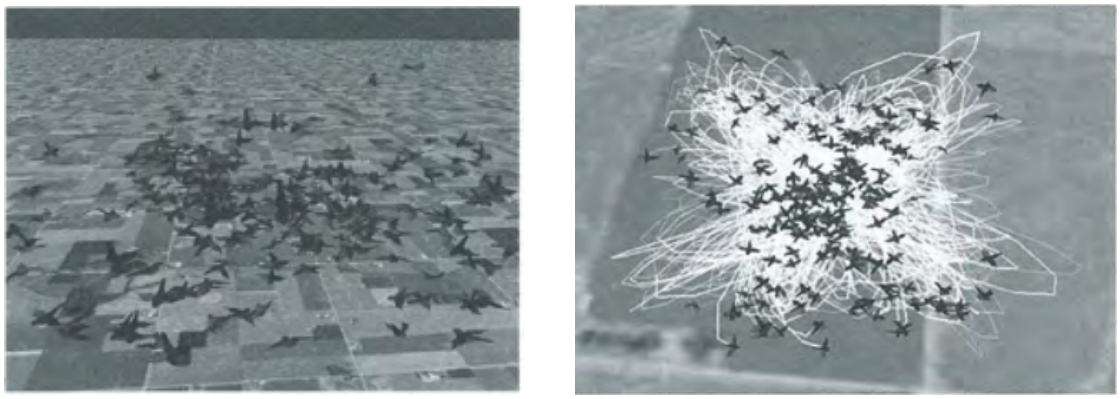


Figure 2.3. Autonomous boids in action (Beneš, 2006).

A shooting off is simulated by constantly increasing the velocity of the chosen boid for a period of time. The results of this method are shown in Figure 2.3. Other extensions of boids include an addition of fitness functions and competition (Reynolds, 1994) and a dynamic adaptation to an environment (Reynolds, 1993).

The work mentioned above shows significant evidence to the fact that particles are an interesting and universal tool. In the case of emerging behavior, particles are capable of producing very complex and sometimes even unexpected results.

2.3 Textures

Texture synthesis is highly researched topic in texture generation research. Seldom reaction diffusion methods are considered a part of texture synthesis but more times than not reaction diffusion systems are looked at separately. For clarity, this review separates them into two separate groups. Texture synthesis itself can be subdivided into many subgroups; this review will reflect on the main relevant subgroups of it.

2.3.1 Reaction diffusion

Reaction diffusion is a method of biological pattern creation. It can be described as two or more chemical substances that diffuse with one each other and create a stable pattern of spots and stripes. In Turks paper (Turk, 1991), the author describes a method of using reaction diffusion to generate textures. This is made possible by creating a mesh over the surface and applying the reaction diffusion method on the mesh. The mesh is represented by evenly distanced points over a model where the adjacency is defined by a Voronoi diagram. An example of a texture generated with reaction-diffusion is shown in Figure 2.4.

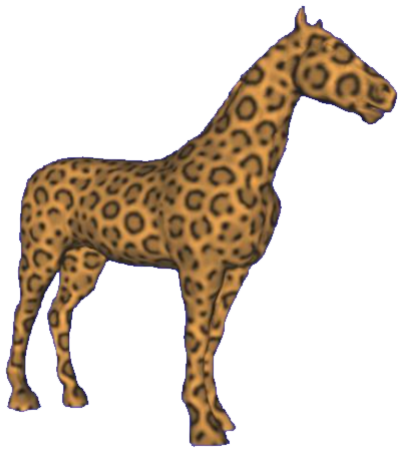


Figure 2.4. A texture generated with a reaction-diffusion method (Turk, 1991).

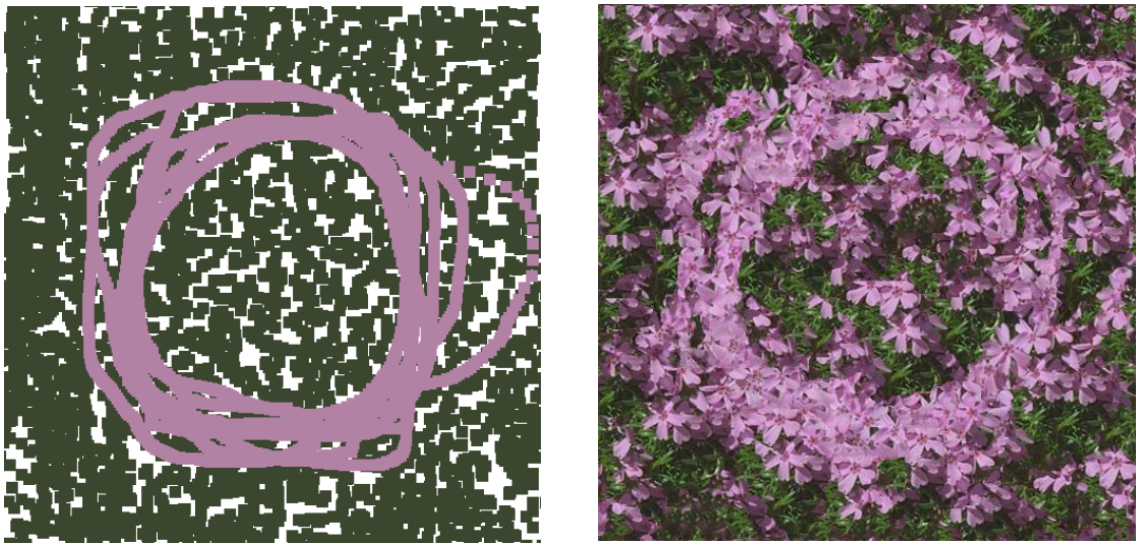
Another paper by (Witkin & Kass, 1991) extends and builds on the traditional reaction diffusion system by allowing spatially, non-uniform, multiple competing ways and anisotropic diffusion. In their conclusion the authors stated that it is impossible to create one universal tool for reaction diffusion; however identifying the processes that are widespread could be highly useful in computer graphics. Although methods of reaction diffusion are gaining popularity in a variety

of fields, the application of reaction diffusion in computer graphics is still in the early stage of development. There is a great amount of room for improvement.

2.3.2 Texture synthesis

(Ashikhmin, 2001) proposed a simple algorithm for texture synthesis. The method enables a user to create repeating patterns of small objects with similar yet irregular size such as flowers, bushes, and tree branches. The way the algorithm works is that the user provides the initial image of an object such as a flower and the algorithm creates a similar set of images to the original image but of irregular size. The algorithm does not change the spatial frequency of the original image or the overall look of the input image. An example of an input and an output of the algorithm are shown in Figure 2.5. The user sets the constraints for how irregular the size can be. The author states that the algorithm performs very fast. The overall system is user friendly and allows for a lot of control utilizing a painting-style interface to paint the objects and edit the properties. In the future work portion of the paper, the author states that this particular paper is narrowed in terms of its focus on particular objects; anything outside of this set of objects is not going to perform well. He also says that even though a set of objects can be greatly extended for this algorithm, it would still be impossible to create a universal tool that can perform well with any kind of object. Opportunity for improvement was also mentioned in the area of textures that have some illumination effects and in the area of algorithm performance optimization.

The next texture synthesizing approach takes the subject up one more notch and introduces texture synthesizing in combination with solid textures. The method was introduced by (Kopf et al., 2007) and it synthesizes solid or 3D textures from 2D sample textures. One of the results of the method is shown in Figure 2.6. The algorithm works by taking a 2D texture and putting three copies of the texture orthogonally, which results in the creation of volume data or voxels. Next voxels are



(a) Input (b) Output

Figure 2.5. Texture synthesis sketch (Ashikhmin, 2001).

optimized by seeing 2D neighborhoods on three orthogonal sides. The voxels create a cube that is filled with texture information at any given point inside of the cube so that when a model is placed inside every part of the model is textured nicely. One of the main advantages of solid textures is that they do not require parameterization of the surface or in other words UV mapping. The UV mapping technique is worse than the solid textures approach because even models with very simple topology are going to have visible seams and distortion. A lot of natural materials can be realistically represented by solid textures. Natural materials such as wood, stone, and marble all are great candidates to be a solid texture.

Furthermore the authors put the additional optimization device into the method in addition to the synthesis method. This helps to avoid the algorithm getting held in the bad local minima or not using the full capacities of the sample 2D textures. The optimization device works by utilizing histogram matching. The histogram makes sure that the solid texture is very close to the sample 2D textures not only locally but globally as well. The histogram matching also improves the

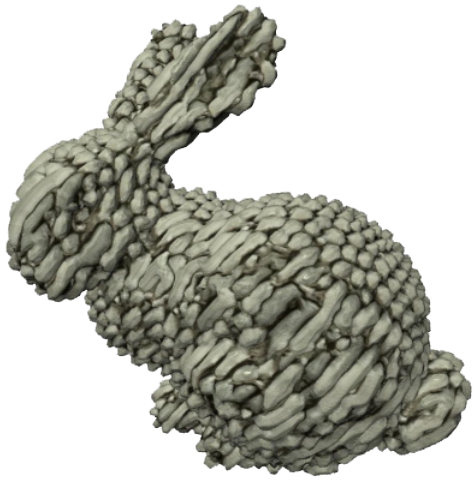


Figure 2.6. An example of a 3D texture (Kopf et al., 2007).

performance of the methods and permits the use of smaller neighborhoods. Once the method is computed, it can be applied to many different models without the need for recomputing it. The method performs well on both the exterior and interior of the model. If the solid texture is moved or rotated, the pattern or texture on the model will move as well. The limitation is that even the algorithm performed well on a reasonable number of 2D textures, there is also a number of textures that it did not do as expected. One of the possible solutions that was proposed by the authors for future work was to include extra slices in optimization, which could solve this problem.

The problem of texture synthesis has also been approached from the performance optimization side (Wei et al., 2008). In their work the authors center mainly on globally varying or homogeneous textures. Furthermore globally varying textures sometimes are associated with a control map. A control map can control some visual attributes of the image. For example if there is an image of crackling paint, a control map could control the thickness of the paint. Many different

research papers use control maps with globally varying textures but other researches sometimes name it differently.

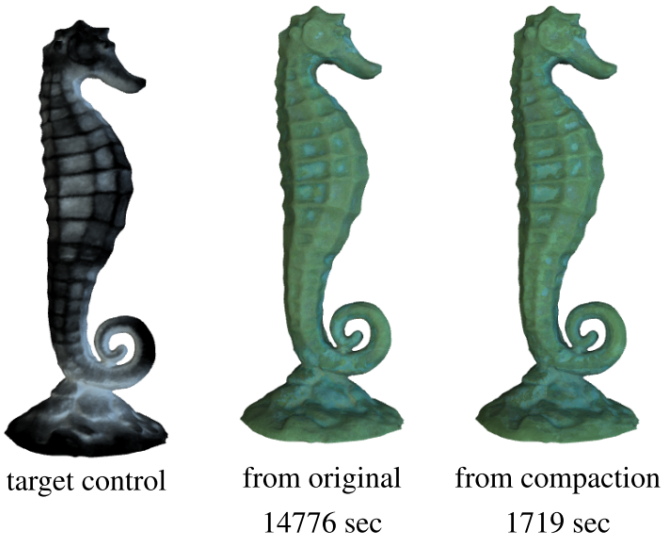


Figure 2.7. Time comparison of a texture synthesis with and without (Wei et al., 2008) method.

(Wei et al., 2008) argue that the results of any texture synthesis are dependent on the size of the 2D sample that is fed into the algorithm. The sample should be as small as possible but contain enough texture information. Keeping this in mind, the authors attempted to go in a different direction; in particular after the algorithm receives a large sized globally varying texture, it goes and creates compact representations that summarize the most important features of the original image. The amount of time the method gains and the quality it produces are illustrated in Figure 2.7. The most important part is that now the texture synthesis can be computed in real time on a GPU. It would not be possible otherwise because there is not enough memory on the GPU to fit the original texture.

According to the authors, it is also possible to go backwards and reconstruct the original large image from the small representations or create a new texture with user supplied control maps. The authors supply a paint-like tool where the user can

paint the texture right on the surface and the brush strokes will effect the control map underneath. The tool works in real time and all the texture information is processed on the GPU. The tool gives a what you see is what you get experience.

There is an ongoing problem with procedural texture generation that takes a few exemplars and tries to generate something in between. It is not trivial to mix features of the exemplars without breaking visual quality. A paper by (Risser, Han, Dahyot, & Grinspun, 2010) describes a method that uses a novel jitter technique that improves the retaining of the structure of the texture after synthesis. The jitter technique works by utilizing a multiscale descriptor. The algorithm attempts to jitter actual features of a texture rather than just pixel values.

Apart from the traditional way of thinking about textures in computers graphics, which is really an attempt to simulate a real texture using a 2D image, the authors (Ma, Wei, & Tong, 2011) try to synthesize textures with 3D objects. They begin by saying that all of the phenomena around us can be thought of as a combination of the same or similar objects such as a statue made of pebbles or a stick house. The authors named these objects discrete element textures. They present a method that can synthesize discrete element textures. The approach is data driven.

The way the algorithm works is first the user provides a sample of the texture to be populated and the desired shape of some size. Next the algorithm goes and assembles the shape as if it was built of the sample objects. For example, the user could supply a set of tree logs and a shape of some sort of house. The algorithm will then go and create something that looks like a stick house. If such an object is needed, this method can save a lot of time as opposed to placing each element manually. One might think that a physically based method could be appropriate here but that is not the case. The authors explicitly mention that the textures that their method produces will not be suitable to be created with a physics simulation method.

The authors of the paper place a strong emphasis on that fact that the method is data driven and by changing the input data they can generate any shape. A user can also control some boundary conditions. For example the shapes that represent texture can be placed on a plate or box, or simply left in a pile. All the boundary conditions can be done from a single sample. In addition, the objects can be set to follow a certain orientation field. The input sample does not have to be a simple object; the system allows for objects to be quite complex and even deformable.

Moreover, the synthesis method is also capable of editing the distribution of objects. If a user selects and changes the input objects, the method will propagate the change to all neighboring patterns that were constructed out of that sample. Besides the position, the user can also change other properties such as orientation and the color of the objects. Properties can be set according to the neighboring objects value, which will result in something similar to a gradient effect. As soon as any of the properties are changed, it will get propagated with a position change through the method previously mentioned above.

2.4 Particles combined with textures

The next set of works will cover topics that attempt to combine two subtopics that were mentioned earlier. In the research paper by (Dischler et al., 2002), the authors attempted to find the most significant characteristics of a texture. After finding the most important characteristics of the texture they extracted them. The authors called these extracted pieces texture particles. Texture particles can be thought of as a summary of the texture or set of pieces from the texture that closely resemble the original image. The method is capable of extracting the texture particles not only from a 2D texture but also from a 3D mesh. After the texture particles were recorded, they can be applied in a paint-like

manner to a different mesh while still preserving the main character of the original texture. The work flow along with a result are shown in Figure 2.8.

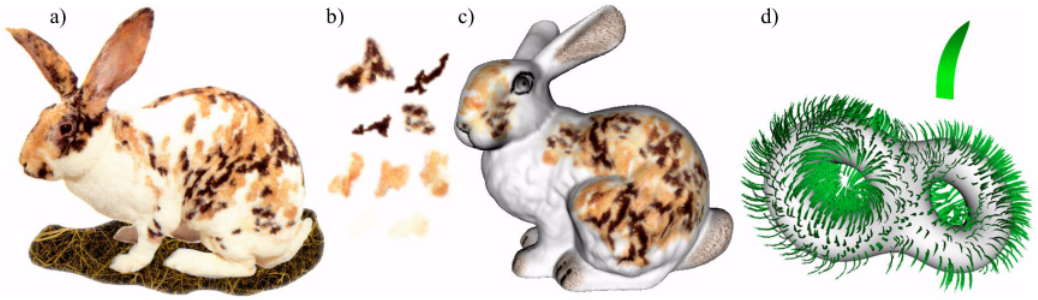


Figure 2.8. a) Input b) Extracted texture c) The texture applied d) The sample texture as 3D model (Dischler et al., 2002)

The main contribution of this paper lies in the synthesis portion because it provides the middle ground between fully automated and manual texture placement. The method tries to combine the best parts of the two. In addition to the method being very fast, it allows for a user to have more control over the synthesis. In particular it allows control over the distribution and sampling of the texture particles. The authors also mentioned a case when the method does not perform well, in particular with textures that represent complex connected spatial arrangements. An example of such texture would be a checkerboard pattern.

The next paper brings the relationship between particles and textures even closer together. The paper authored by (Fleischer, Laidlaw, Currin, & Barr, 1995) tries to approach the problem of creating small details such as scales, thorns, and feathers by utilizing a particle system. The core of the method uses a particle system in combination with development models and reaction-diffusion methods to create various kinds of features. The system is controlled by a set of parameters that help to create the features needed. Because the method uses particles to create the texture, no stretching or shrinking is introduced, which is known to be a common problem for the traditional approaches such as texture mapping or bump mapping.

The system also showed positive results on models with unusual topologies. An example of a model with an unusual topology is a knot. The system is also size adaptive in relation to the corresponding polygons. The smaller the polygons underneath, the smaller the feature is going to be. The distribution and orientation of cells are controlled through cell division and initial seeds. One of the limitations the authors mentioned is that the conversion from particles to textures on more complex models sometimes produced unexpected results. In addition, the simulation speed varies greatly and can take from a couple of seconds to a number of hours due to the data computed from the particles before texture creation being very large.

Particles can be also combined with textures by means of attaching a texture to a particle. The texture uses a particle as transfer mechanism to get around a scene. This technique is usually used as part of another method called billboarding. If a texture is not to be mapped on a flat orthogonal to the camera surface this is usually referred to as a decal. (de Groot, Wyvill, Barthe, Nasri, & Lalonde, 2014) describes a method of projecting many repeating decals on a 3D surface. The method makes decals deform each other as they compete for space. Another method that deals with applying a decal to a complex geometry is described by (Lefebvre, Hornus, & Neyret, 2005). In their work, the authors apply high resolution texture to a surface without any global planar parameterization. This is done by storing the attributes of the decal in 3D hierarchical structure. The structure surrounds the mesh and can be dynamically changed.

2.5 Summary

This chapter provided a review of the literature relevant to particle systems and textures. After reviewing a number of research works mentioned above on a variety of particle system topics as well as on texture generation topics, it is evident that many limitations exists. It was pointed out in the future work section of several research works that particle systems can be extended to perform as a texture

and to be more easily controllable. The next chapter provides the framework and methodology to be used in the research project.

CHAPTER 3. FRAMEWORK AND METHODOLOGY

This chapter provides a detailed explanation of the framework and methodology used in the research study. In particular it describes the types of input and output, simulation engine, and extensions that were made to enhance it. In addition, a data structure used in the engine as well as a 2D framework are presented as a part of this section.

3.1 Overview

This study focuses on the creation of a particle system framework that extends (Reynolds, 1987) model for creation of a specific class of textures and image effects. The framework is comprised of features and controls that allow for a generation of a large range of different textures. The framework works independently of the 3D model resolution as well as texture coordinates.

The framework uses a 3D mesh and images as inputs. The boid engine receives the inputs, applies, and process them. The user can manipulate a set of parameters to adjust the simulation. The output of the framework is boids with or without decals attached to them that create a pattern or a design on the surface or canvas. An overview schematic is shown in Figure 3.1.

3.2 Input & output

The framework uses a 3D mesh as an input. The mesh is used as an object to which a simulation will be applied. A user may choose to upload a number of images to the framework, which will serve as decals or building blocks of a texture. Images can also be given as an input for vector field generation. After the images

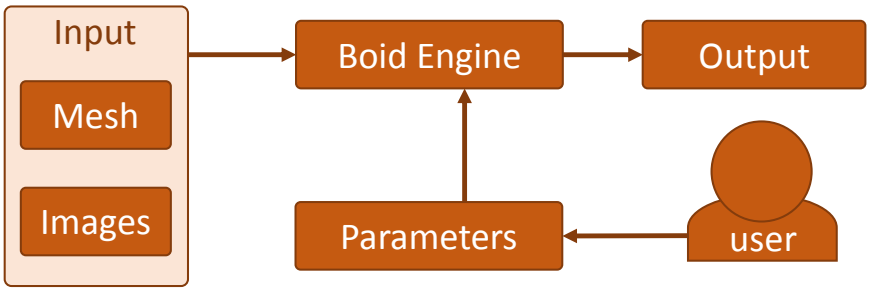


Figure 3.1. Overview of the framework

are in place the user may begin to populate the surface of the model with particles. The particles that have an effect on each other or are allowed to have interactions are arranged into particle layers. All particles of one layer are completely independent and not influenced by particles in the other layers. The framework provides a separate control for each boids' layer. A 3D work flow of the framework is shown in Figure 3.2.

In order to improve the user's control of the texture generation, the framework is equipped with visualization capabilities. These capabilities allow a user to see influence regions or "the reach" of every force of every boid. In addition, the visualization of velocity vector and projected velocity vector, in case of the mesh not being planer, is also present. The framework has a few different ways to visualize the imported mesh. The user may choose between wireframe, flat shading, and Phong shading modes.

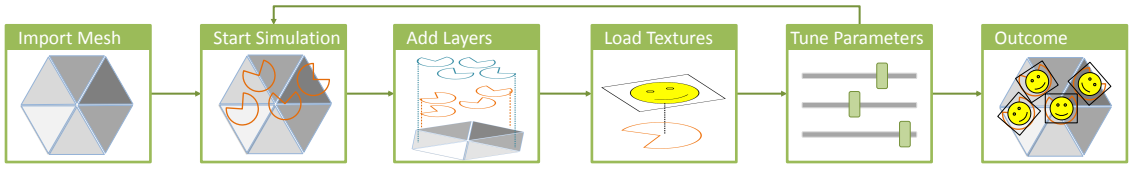


Figure 3.2. Interaction work flow

3.2.1 Texture element separation

The framework that is presented in this study attempts to generate a particular class of textures. Namely, this class is comprised of textures that can be separated or segregated into layers. Each layer in the texture carries a particular feature of that texture. The features can differ from one to another by size, color, or both. When all the layers are put together, a more complex multiple feature texture emerges.

Examples of such textures can be found in many places around us. This textures appear on concrete facades of buildings, linoleum floor patterns and carpets. Thus the simulation of such textures will be useful. Four examples of the textures that belong to the class of textures whose features could be segregated are shown in Figure 3.4.

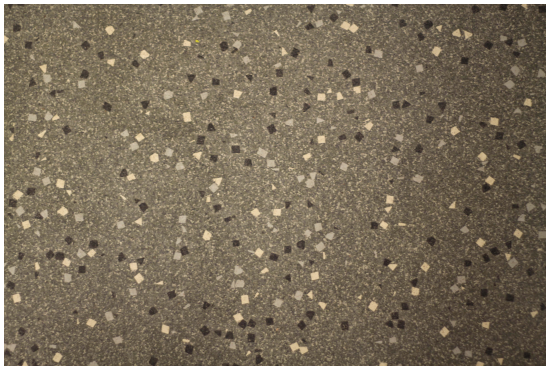
Figure 3.3. An example of a texture that can be segregated into layers

For example if we look at Figure 3.3 this texture can be segregated in the following ways: The first layer is the base layer, which can be represented by just a solid color or a noise texture to fill in small details. All consecutive layers have the

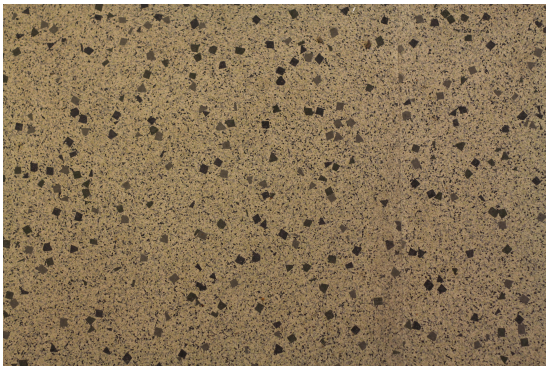
appropriate feature assigned to them. In this case all the features in the texture have about the same size so the layers will be segregated by color. The second layer will have white features, layer three will have black features, and layer four will have yellow features. So the texture can be thought of as a composition of these four layers.

(a) Example 1

(b) Example 2



(c) Example 3



(d) Example 4

Figure 3.4. Additional examples of textures that can be separated into layers

3.2.2 Textures with Boids

Textures can be separated into layers based on their similar features. There are many characteristics that can be observed about each feature such as color, size,

and how often each feature appears throughout the texture as well as how far apart similar features are. The first two characteristics are inherit to the image but the last two can be simulated with boids. Boid simulation will have a feature attached as a decal to every boid that is part of the feature layer. By putting each feature group on a separate simulation layer, a collage of the features with different characteristics will be created. A break down of features into layers is shown in Figure 3.5. Each layer is completely independent and unaware of others. Different configurations of passing an image to a boid layer are possible. Every boid does not have to carry an image in a boid layer. A boid layer might have a sparse decal assignment. Boids without decals still effect movement of those that do have a decal attached to them. This will create more irregularities in pattern.

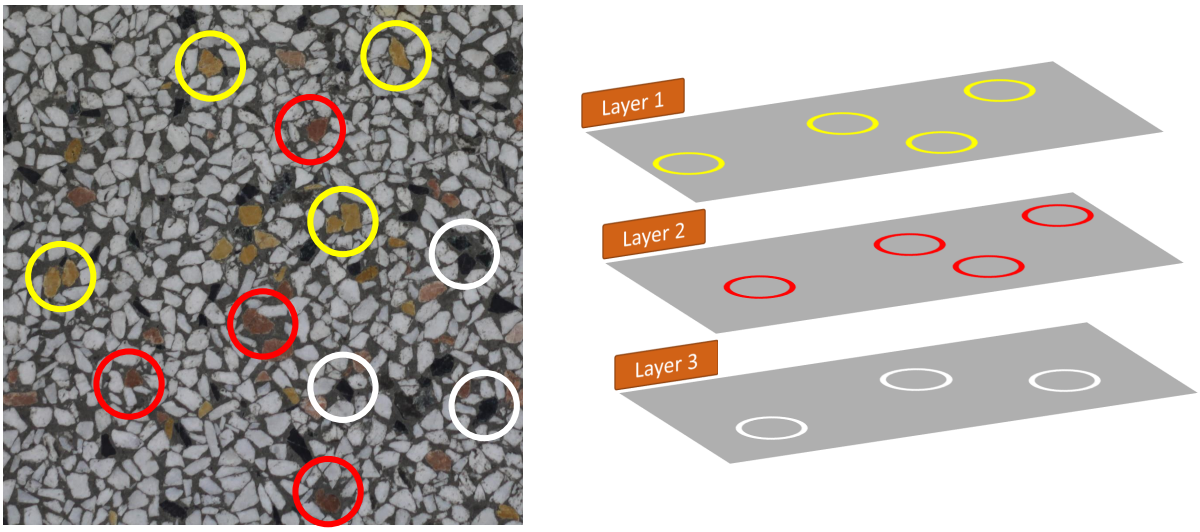


Figure 3.5. Separation of features into boid layers.

3.3 Classic Boids

This section will cover in detail (Reynolds, 1987) Boids simulation as well as new features that were added to it to improve texture generation. The simulation is

the main engine behind the framework. In addition, the section describes other techniques that were attempted in the process of creating the framework.

3.3.1 Rules

Reynolds boids simulation is a particle simulation, which allows for emerging behavior to occur. Emerging behavior is a behavior that emerges from a multitude of agents in the simulation. Any single agent is not aware of any global variables in the simulation. Every agent has a limited set of rules, based on which it determines its actions. A particular set of actions for the agent may vary from different kinds of simulations. Collectively all agents by acting on their set of actions allow for a higher order of behavior to occur. During the simulation, it appears that the crowd of unconnected agents act as it was one coherent unit instead of a number of disconnected agents. Reynolds boids simulation is one of many simulations that allows for emerging behavior to occur.

The way the simulation works is that every agent in the collection has three rules or forces as they referred to in the framework. The forces are separation, alignment, and cohesion. In the framework each agent starts out with some velocity and a position. After every iteration, velocity of each agent is recomputed and integrated for the next step. The computation of the new velocity involves the summation of the current velocity of the agent with the newly computed separation, alignment, and cohesion vectors.

All three forces share a common feature. The result vector from any of the three forces is computed based on velocity vectors, positions and number of boids within a visibility circle of a boid that is being computed. The boid that is being computed for simplification will be referenced as a current boid. The visibility circle of a boid is referred in the framework as a radius of a boid. In order to determine if a boid has any other boids within its radius, a Euclidean distance is computed to every boid on every iteration. If there are no boids found in the radius of the

current boids, a zero vector is returned, which means the velocity of a current boid will not be changed.

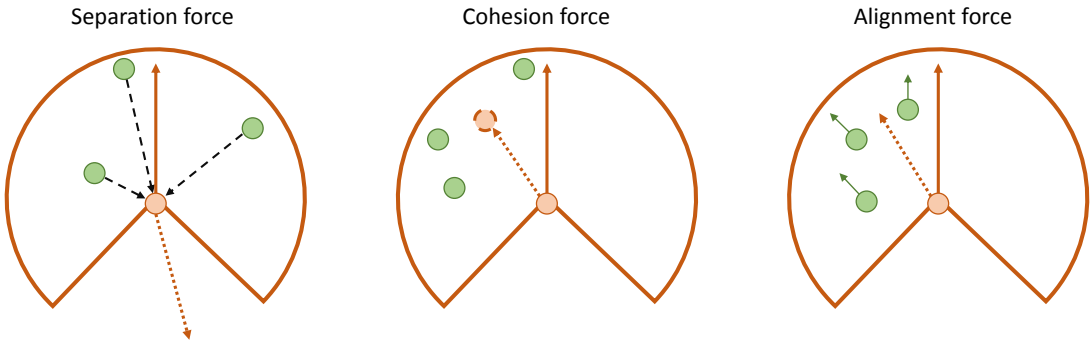


Figure 3.6. Reynolds boids forces.

Separation force: After the boids that are within the radius have been determined, the difference vector is computed. The difference vector is computed for every boid within the radius by subtracting its position from the position of the current boid. The result is a vector that points from the subtracted boid's position to the current boid's position as shown in Figure 3.6. All the difference vectors are summed into one vector and divided by the number of boids within radius of a current boid to get an average vector. This average vector is the separation vector. The separation vector is represented by a dashed arrow in Figure 3.6. The purpose of the separation force is to keep boids from clustering into one tight group by forcing them apart. The calculation of separation force is summarized in the Equation 3.1.

$$\vec{s} = -\frac{1}{|V_i|} \sum_{\forall b_j \in V_i} b_i.pos - b_j.pos \tag{3.1}$$

V_i stands for a set that contains all the boids withing the radius of the current boid and *pos* stands for boid's position. Every boid contains its own position and velocity. In all following equations velocity and postilions are referenced in similar manner.

Cohesion force: This force is concerned with the positions of boids within the radius of the current boid. The positions are summed up and divided by a number

of boids within the radius. This results into an average position or a centroid. The centroid is shown in Figure 3.6 as an orange circle with a dashed stroke. Lastly, the position of the current boid is subtracted from the centroid. This results in a vector that points from the current position to the centroid. This vector is the cohesion vector. The resulting cohesion vector is shown with a dashed arrow in Figure 3.6. The purpose of the cohesion force is to counterbalance the separation force by steering boids to the center of a flock. The calculation of cohesion force is summarized in the Equation 3.2.

$$\vec{c} = -b_i.pos + \frac{1}{|V_i|} \sum_{\forall b_j \in V_i} b_j.pos \tag{3.2}$$

Alignment force: This force is concerned with velocities of boids within the radius of the current boid. The velocities are summed up and divided by a number of boids within the radius. This results into an average velocity. This average velocity is the alignment force. It is shown in the Figure 3.6 by a dashed line. The purpose of this force is to create uniform velocity, so the boids move as a flock. The calculation of alignment force is summarized in the Equation 3.3.

$$\vec{a} = \frac{1}{|V_i|} \sum_{\forall b_j \in V_i} b_j.vel \tag{3.3}$$

3.3.2 Extensions

We have extended the classic boids simulation. The extension consists of new features that are added to enhance the movement of boids. In particular, features include new calculation of forces and a combination of boids with a vector field. These features are a part of the contribution of this thesis. The features help to enhance the simulation by providing a more realistic approach. These extensions are covered in detail below.

Front vision: This feature allows for a current boid to concentrate on not only boids that are within its radius but also in front of it. The exact degree of

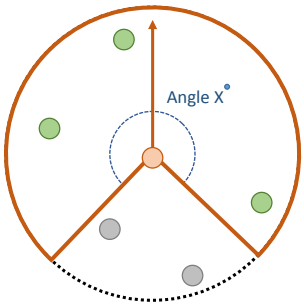


Figure 3.7. Front vision. Black boids are discarded from the calculation of forces.

visibility is a parameter in the framework and can be adjusted on the fly. It can vary from a very narrow visibility to a full circle visibility. This simulates as a boid has an actual vision and cannot see behind it, thus should not be influenced by boids it cannot see. The angle is calculated by first finding a vector that points from a current boid's position to a visible boid's position. Next, an angle between the resulted vector from the previous operation and the boid's velocity is computed. If the angle is less than a parameter, the boid will participate in the computations of the forces listed above, otherwise it is discarded. Figure 3.7 shows a black dashed line where the circle of visibility could have been if this feature was not there. It also shows that boids that are marked black are discarded because they are not within the radius of the current boid. The calculation of the front vision is summarized in the Equation 3.4 where B is a set of all boids, r is a visibility radius and *alpha* is angle of vision.

$$V_j = \{b_i \in B | \forall b_j \in B | b_i - b_j | < rand | b_i \measuredangle b_j | < \alpha\} \tag{3.4}$$

Acceleration based on a scale factor: This feature works close in hand with the all three classical forces listed above. In particular, the feature puts an importance on the distance between the current boid and a boid within its radius. The idea is that the boid that is closer to the current boid should influence the

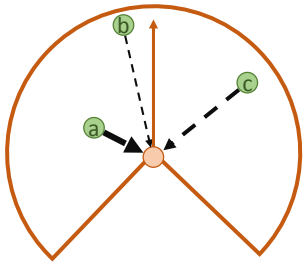


Figure 3.8. Acceleration based on a scale factor.

current boid's forces stronger than the one that is barely on the edge of the visibility radius. In a sense this mimics a real situation when a driver drives a car and has to react to an immediate danger quickly as opposed to if the danger was far away. This idea is shown in Figure 3.8 with dashed black lines, the thicker the line the more influence this boid should have on the current boid.

The way this force gets computed is by first finding a centroid position of visible boids. This step is similar to the centroid computation in cohesion force. Next, a multiplier vector, using the current boid's position to the centroid, is computed. The length of the multiplier vector will be used to divide the radius of visibility radius to get a factor. The corresponding force will be multiplied by the factor to get the final result. If the length of the multiplier vector will be equal to the radius, the factor will be at its minimum, which is one. If the length is anywhere in between zero and the radius, the factor will be greater than one. The factor of one will have no impact on a force but if it is greater than one, the factor will cause a force to accelerate. The calculation of the acceleration factor is summarized in the Equations 3.5 and 3.6 where r is a visibility radius and C is centroid position.

$$f = \frac{r}{||C - b_i.pos||}, \tag{3.5}$$

where

$$C = \frac{1}{|V_i|} \sum_{\forall b_j \in V_i} b_j.pos \tag{3.6}$$

Boids with vector fields: The vector framework is a 2D boids simulation. It was created as a 2D framework because it allows for trivial mapping to 2D image space. The vector framework primarily focused on the application of the vector field as an additional force to the classic boids. The movement of the boids in this framework is traced in various ways and produces a number of different textures and image effects. In addition, other techniques were tested with this boids simulation. The techniques include color manipulation, Voronoi diagram application, and color clustering. The main purpose of these techniques were to explore different visual qualities.

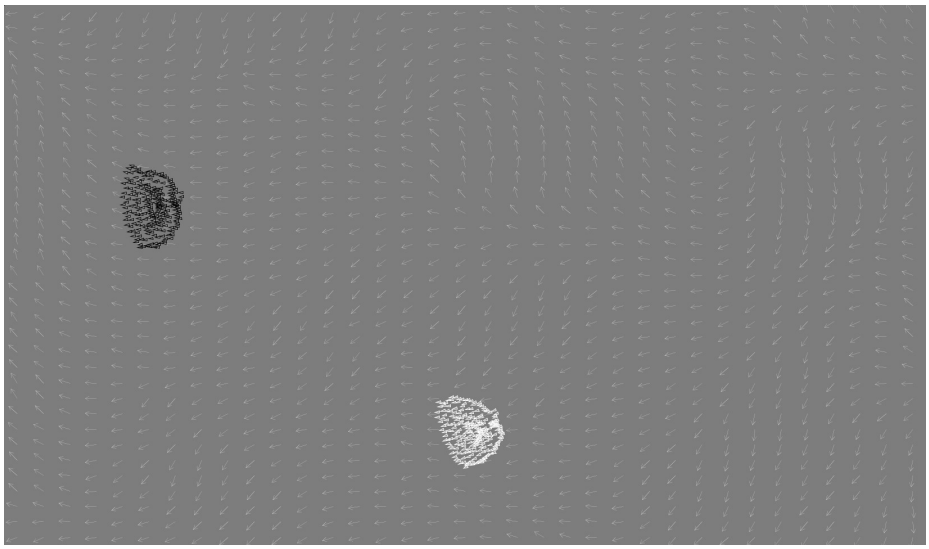


Figure 3.9. Vector field generated via procedural function.

In the vector framework, a vector that is coming from a vector field is counted as another force navigating the boid. The final vector that is added to the current boid's velocity consists of a sum of separation, cohesion, and alignment vectors plus a vector obtained from a vector field. A screen space is divided into a grid of cells, each cell contains a vector from a vector field. Whenever a boid moves, it queries based on its position which cell is it in and receives the appropriate vector. A pseudo code for this calculation is shown in 3.11. The vector framework

allows for the resolution of the vector field to vary. There are three ways to generate a vector field within the vector field framework. One way is via procedural function. An example of procedurally generated vector shown in Figure 3.9. The second way is by using an image and extracting a gradient from it. There is also a third way that is supported by the vector framework, which is very similar to the second. The only difference is that the image is drawn by a user.

```
1:  procedure GRADIENT
2:      j ← 0
3:      j ← 0
4:      while  i < image.width() do:
5:          while j < image.height() do
6:              vector.x ← image[i + 1][j] − image[i][j].
7:              vector.y ← image[i][j + 1] − image[i][j].
8:              vectorField[i][j] ← vector.
9:              j ← j + 1.
10:         end while
11:         i ← i + 1.
12:     end while
13: end procedure
```

Figure 3.10. The algorithm that describes a generation of a vector field from an image.

There are a number of ways to extract a gradient from an image. The vector framework does it by iterating through all the pixels in the image and subtracting the current pixel value from a pixel right in front of it and right below it. A pseudo code for this calculation is shown in Figure 3.10. If the vector field grid has smaller dimensions than an image, the step to get to the right pixel and to the pixel below is scaled by a constant. The two results from the subtraction create a 2D vector. In

order to get a better vector field with less noise, the input image is changed into a black and white image. Every pixel is either complete white or complete black with no gray values. This step not only creates an extreme gradient on the edges of the image but also creates large regions of a solid color. A solid color region has no gradient and thus no noise. Next, a black and white image is fed for vector field generation. The resulting vector field is shown in Figure 3.12. After the vector field has been generated, boids are used to trace their way as they travel around 2D space. The vector framework has a parameter that sets the degree by which the boid's velocity is influenced by the vector field.

```
1:  procedure BOID MOVE
2:      i ← 0
3:      while  i < boids.count() do:
4:          separation ← getSeparation(boids[i]).
5:          cohesion ← getCohesion(boids[i]).
6:          alignment ← getAlignment(boids[i]).
7:          position ← boids[i].getPosition().
8:          vector ← vectorField.getVector(position).
9:          velocity ← separation + cohesion + alignment + vector.
10:         boids[i].SetVelocity(velocity + boids[i].GetVelocity()).
11:         i ← i + 1.
12:     end while
13: end procedure
```

Figure 3.11. The algorithm that describes movement of boids.

Figure 3.12. Vector field produced from the black and white image.

3.4 3D mapping

The classic 3D boids, the principles of which were described above, travel though 3D space unconstrained. In order to make them form a pattern or texture on a surface 3D mesh, some type of mapping their movement to a surface has to be

designed. This section describes in detail a technique involved that constrains the movement of boids to the surface of a mesh.

3.4.1 Initial approach

There are two properties of a boid that need to be modified for it to be mapped to the mesh. These are the boid's position and velocity. The position cannot be anywhere outside the surface. The velocity defines the next position, so it has be parallel to a surface. The initial idea was to constrain the boid's positions to only vertexes of the mesh. The boids would move as they did before in 3D and only after the final position was computed, another function would compute the closest vertex to the current position of a boid and put the boid there. There are a number of problems with this approach; it is slow and if the mesh was coarse the steps might be very large to mention a few. It is slow because after every step each boid have to find the closest vertex on the mesh where to move to, which requires iteration through all vertexes. The boids have to have a large step distance because they could not travel in between vertexes. On the positive side this approach requires very minimal changes for position calculation and no changes for velocity calculation.

3.4.2 More refined approach

In order to decrease step size and make boids move smoother, a better approach was needed. The boids had to be able to move within a single triangle and then smoothly transfer into the next triangle in front of it. The movement within a triangle is fairly easy and similar to moving on a surface of a plane. The only problem is that the velocity of the boid will probably not be parallel to the surface. This means the boid will detach from the surface after very few steps. The problem was solved by projecting the velocity vector of the boid on the current surface or on the triangle it is on. The next position of a boid is calculated based on the projected

velocity. This ensures that the boid always moves on the surface of the mesh. As far as the boid is concerned, it still moves in 3D space "unaware" that it gets mapped to a surface. The projection of the velocity is illustrated in Figure 3.13. The orange arrow is the boid's velocity and the dashed blue arrow is the projected velocity.

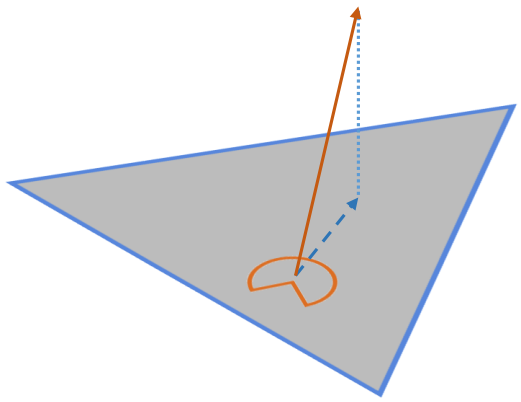


Figure 3.13. Projection of boid's velocity on a triangle.

3.4.3 Data structure

The previous subsection described the movement of a boid within a triangle; this subsection describes what happens when a boids reaches an edge. As it was discussed above the initial approach had to iterate thought all the vertexes to find the next position. Essentially, a similar process could have been used with triangles as well. In this case, we would look not for a closest vertex but for a triangle that has the same edge that the boid is crossing within the current triangle. Needless to say, iterating thorough all edges of all triangles will be slow. A better technique had to be found. One way to solve this problem is to create a precomputed index table data structure. Precomputed would be mean that the data structure would be created when a mesh is loaded to the framework. It would have to be created only once unless the mesh is changed. The data structure stores four things in each cell.

The first entry in a cell is an index of a triangle. All the triangle indexes are loaded into the data structure. This means that the number of triangles in a mesh is equal to the number of cells in the data structure. The rest of the entries in each cell are filled with indexes of triangles that are neighbors of a triangle whose index is listed in the first entry of each cell. Because the mesh is constructed of triangles and a triangle has three edges, each triangle will have three neighbors. This process of filling the cells of the data structure is illustrated in Figure 3.14.

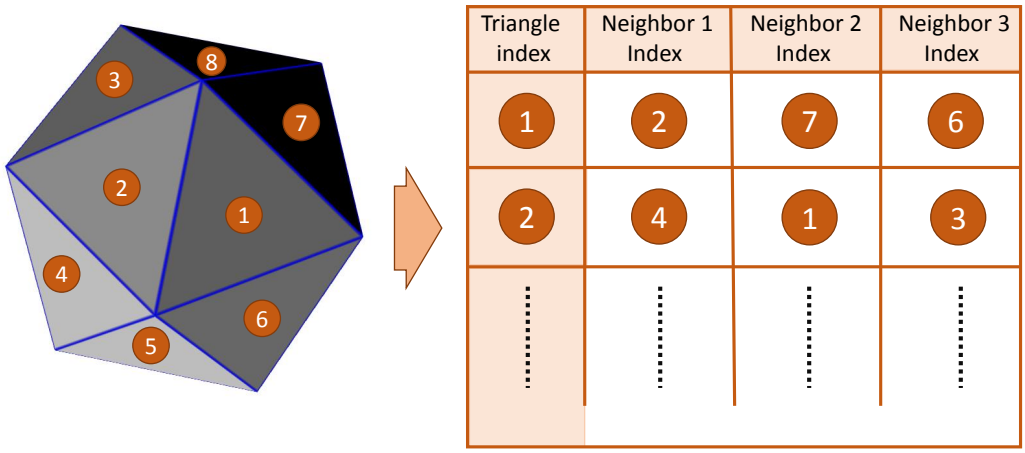


Figure 3.14. The lookup data structure.

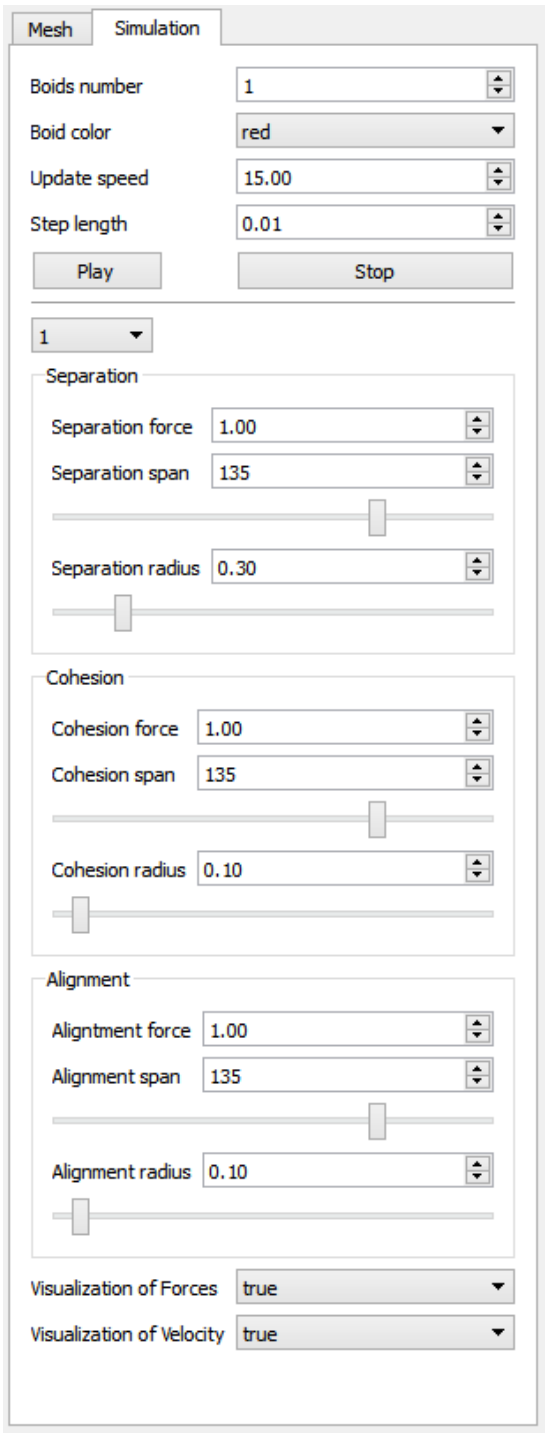
During the simulation each boid keeps track of which triangle it is currently in. When a boid encounters an edge it queries its current triangle index and based on that it can query neighboring triangles. The data structure reduces the search of a common edge from all the edges of all triangles to only three neighboring triangles. The movement of the boid is a recursive step if the boid reaches an edge without making the full step. The move is executed again recursively on the remainder of a step in the next triangle.

3.5 Interactive control

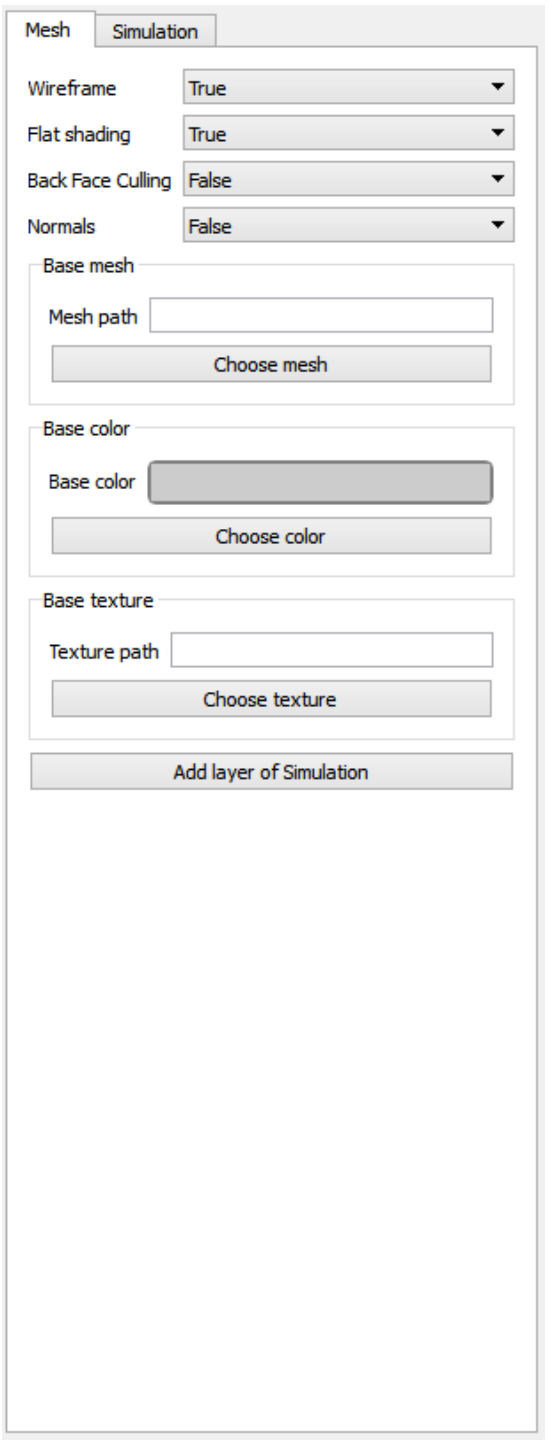
The framework is equipped with a graphical user interface that allows the user to change parameters interactively. The framework's GUI is divided into two sections. Section one controls mesh related properties such as the shading mode, color of a mesh, texture of a mesh, back face culling, and normals. Section two controls the behavior of a single boid layer. The framework has functionality to reuse the same GUI for different layers but only one at a time. The boid layer functionality can adjust the speed of a boid simulation, the step size as well as the radius of visibility and the angle of vision. In addition, each force has a multiplier factor attached, which allows the user to increase or decrease the amount of strength of a certain force. The user can control the multiplier values by moving the sliders. This feature allows to fine-tune a balance between the forces to get the desired result. Both GUI sections are shown in Figure 3.15. The framework also provides a way to visualize the regions of influence of different forces as well as velocity and projected velocity vectors.

3.6 Summary

This chapter provides the framework and methodology used in the research study. It covers the framework overview, engine, and its key components as well as the extensions added. Detailed conceptual as well as mathematical explanations fortify the approach taken. The section defines a class of textures that can be generated with boids simulation. The section also shows the refinement and development of ideas throughout the research study. The next chapter provides implementation details of the framework.



(a) Boid controls



(b) Mesh controls

Figure 3.15. GUI for mesh and simulation.

CHAPTER 4. IMPLEMENTATION

4.1 Platform

The framework was programmed in C++. All the visualization is done with OpenGL. Results were obtained on a Intel Core i7 CPU with clock speed of 3.40 GHz and with 16 GB of RAM memory. The framework was implemented with a Visual Studio 2013 using Windows 8.1. GLM math library was used for all the mathematical computations. The graphical user interface was implemented with QT framework.

4.2 Decals generation

The decals are the images that are attached to the boids. Boids' positions and velocities are generated and stored on the CPU. The mesh in contrast, is loaded to the GPU and then repeatedly drawn to the screen. The framework draws its textures in a fragment shader, which is on the GPU. To make the decals follow the boids, the boid's positions had to be transferred to the GPU in a separate copied data structure. The data structure is passed to a geometry shader. The geometry shader has the capability to create an extra mesh. For every passed position of a boid, the geometry shader generates two triangles or a quad with texture coordinates. The quad is passed to the fragment shader where a decal is attached to it, which allows the texture to follow a boid.

CHAPTER 5. RESULTS

In this chapter the results of the study are described. The chapter is divided to show the results of 2D boids as well as 3D boids.

5.1 2D Boids

5.1.1 Line tracing

The movement of the 2D boids is traced in various ways and produces a number of different textures and image effects. Initially, boids tracing was done without a vector field applied. The textures were created by creating two flocks of boids. The separation or repulsion was set higher between the flocks than it normally would be within the flock. This was done to make sure the flocks do not intermix. A different tracing color was assigned to each flock. This would ensure that the color will be balanced with some other color instead of a screen becoming just one solid color after an extended period of time. The result of tracing boids can be compared as if it was drawn without lifting a pen or as a continuous line. The results of this tracing are shown in Figure 5.1. The pattern resembles a 3D look of a terrain but the individual lines of tracing can still be seen. Figure 5.2 shows the same run but after a couple of hours. After about two hours of simulation, the individual trace lines are a lot less prominent. The textures looks much more balanced and natural. The pattern resembles a look of a rough surface such as dried soil or weathered metal. An argument can be made that boids would be more suitable for making rough surfaces than some procedural noise generation due to the boids' nature. Boids are a lot more regular in their movement pattern, because of

their forces, than noise but element of randomness is still present. This setup was also tested with colors other than black and white assigned to each flock. Figure 5.3 shows these results. Figure 5.4 shows a time-lapse between different changes in texture as the time goes by.

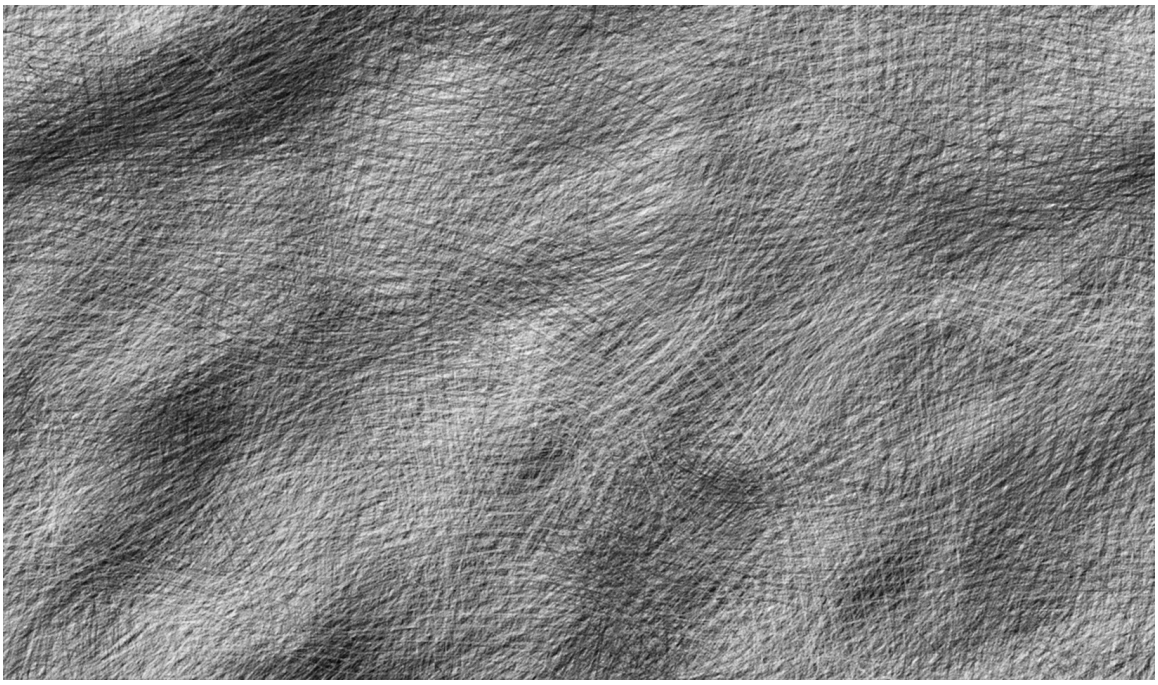


Figure 5.1. Black and white flock are traced (after about 5 minutes of simulation).

5.1.2 Vector field tracing

In this section the results of combining boids with a vector field are presented. This section builds upon what was already stated in previous section. As it was described in the methodology, the framework can generate a vector field procedurally and from an image. An example of a procedurally generated vector field and a short snapshot of a simulation are shown in Figure 5.5. It can be clearly observed that the boids follow the vector field pretty closely. For the vector field

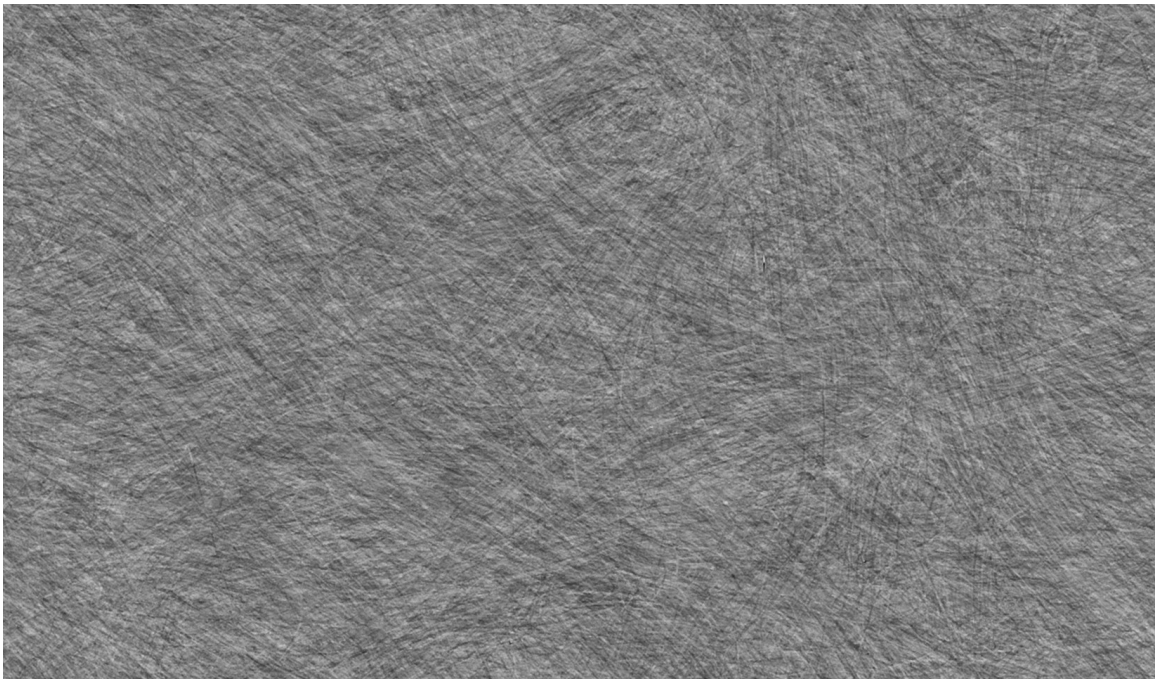
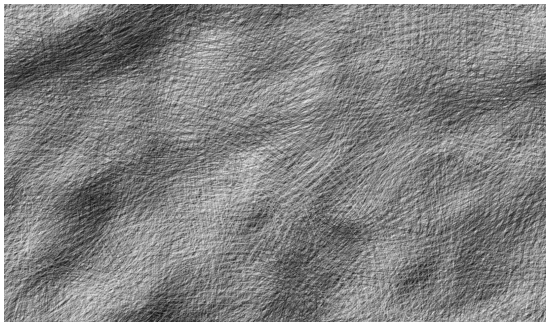


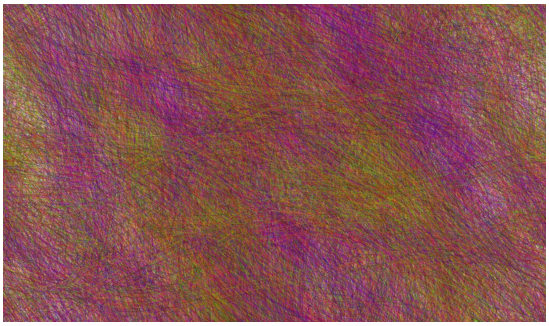
Figure 5.2. Black and white flock are traced (after about 2 hours of simulation).

simulation there is only one flock used with a single black tracing color. There is a far less chance of a screen becoming a solid color because some places in a vector field will be a lot less likely to get visited by boids due to the characteristics of an image. Thus, there is no need for color balancing. A great care is taken to prepare the images for vector field generation. The preparation of images was described in the methodology chapter. A user can adjust settings that control how much the vector field can influence boid's velocity. Figure 5.6 shows the results of tracing the same vector field at different settings.

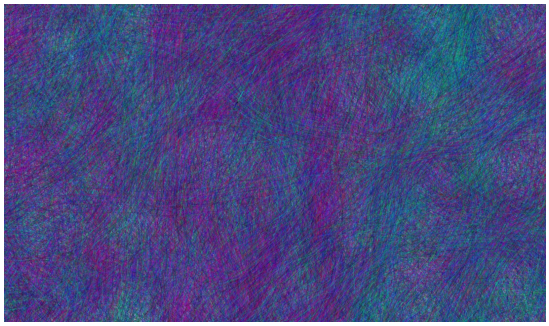
In addition, a user can draw any shape in a 2D canvas within the framework and covert the resulting image into a vector field. The framework provides an interface for choosing a brush size, color of brush, and gives the ability to apply the Gaussian blur to smooth the gradient. Figure 5.8 shows a user drawn image and a vector field created from it.



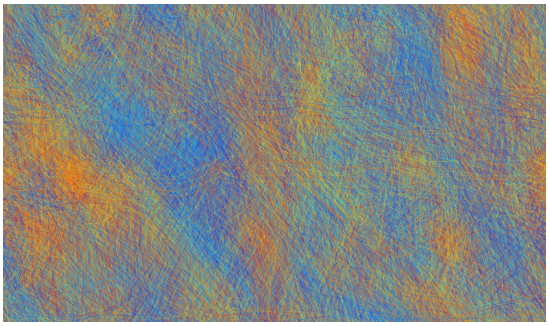
(a) Example 1



(b) Example 2



(c) Example 3



(d) Example 4

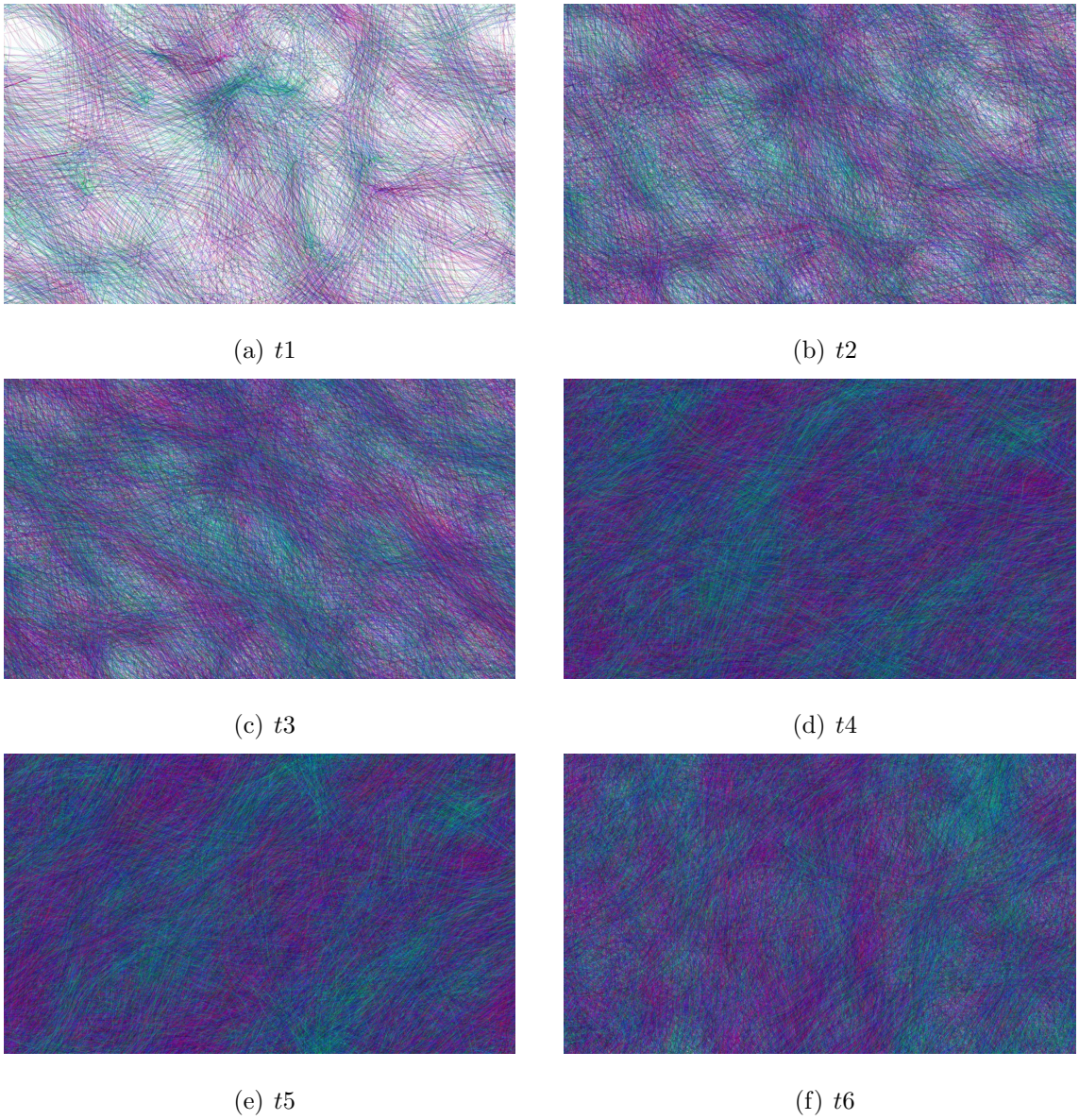
Figure 5.3. Results produced by tracing boids without image input. Two flocks with different tracing colors.

5.1.3 Initial experiments

In addition, other techniques were tested with boids simulation. The techniques include color manipulation, Voronoi diagram application, and color clustering. The main purpose of these techniques were to explore different visual qualities. The result of these experiments are shown in Figure 5.9.

5.2 3D boids

Figure 5.10 shows how decals can be attached to boids. Boids are restricted to the area of the triangle in this simulation. Although there is only one layer shown, there can be many layers of boids carrying features at once. Figure 5.10



(a) $t1$ (b) $t2$

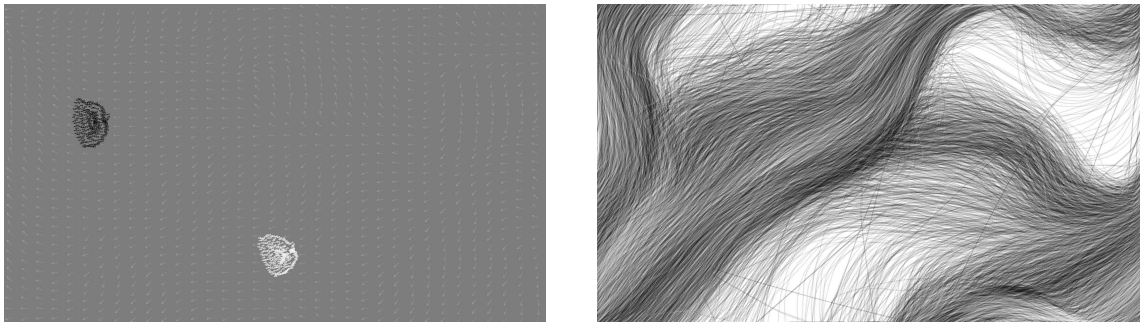
(c) $t3$ (d) $t4$

(e) $t5$ (f) $t6$

Figure 5.4. Results produced by tracing boids without image input. A time-lapse from $t1$ though $t6$.

shows a visualization of forces. By using the GUI, a user can easily change the settings of the forces causing boids to reform.

Figure 5.11 shows the mapping of boids to a 3D surface. The boids are attached to a surface using the data structure and projection described in the

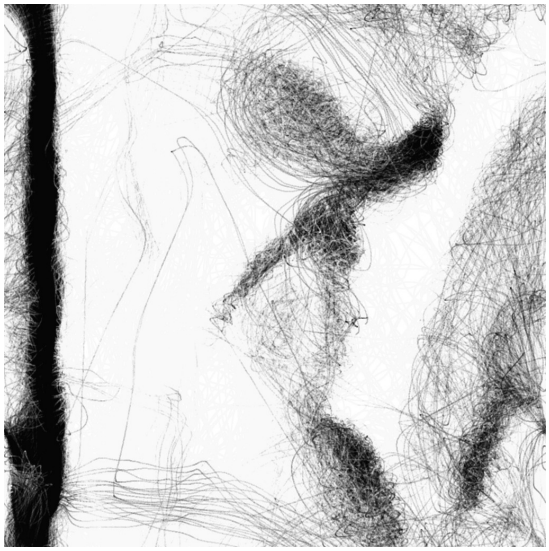


(a) Procedural vector field

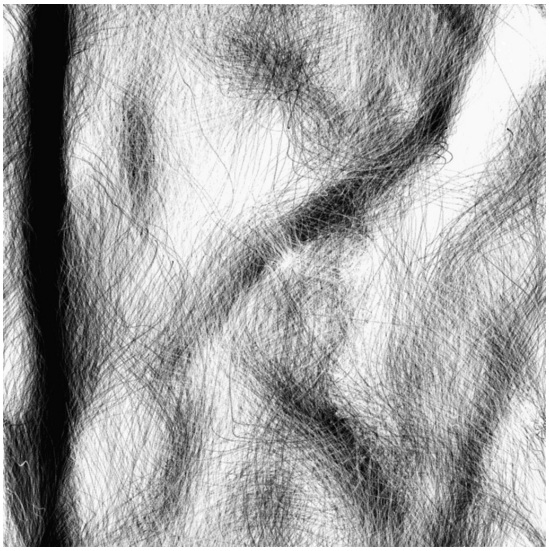
(b) Resulting simulation

Figure 5.5. Boids with a vector field influence.

methodology section. Although the distance between the boids is computed as Euclidean distance, the boids move strictly on a surface. This is done by projecting the current velocity of a boid to a surface before the boid has a chance to make a step along the velocity. This allows a boid to have a velocity not parallel to the surface.



(a) Example 1



(b) Example 2

(c) Example 3

(d) Example 4

Figure 5.6. Results produced by tracing boids with image input at various settings.

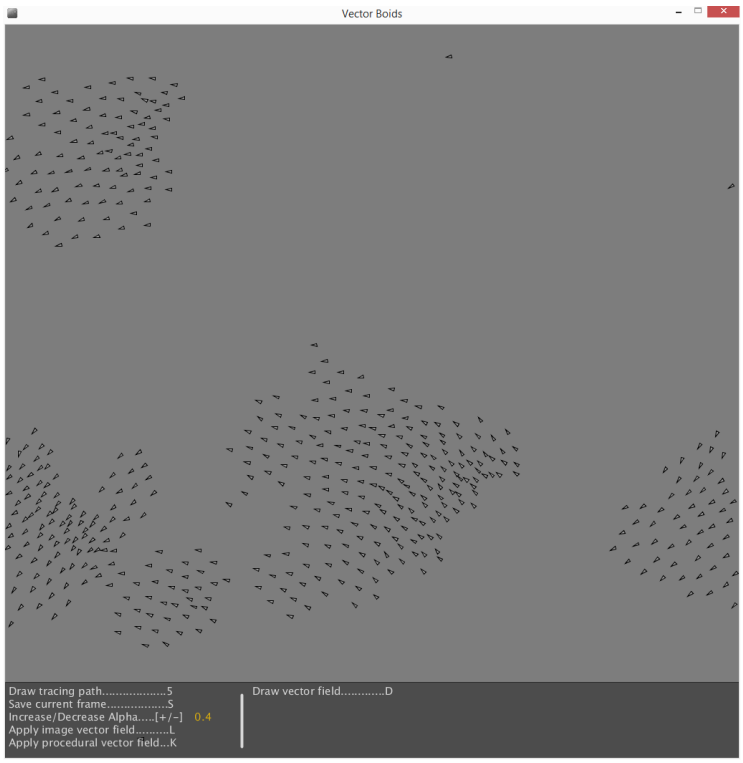
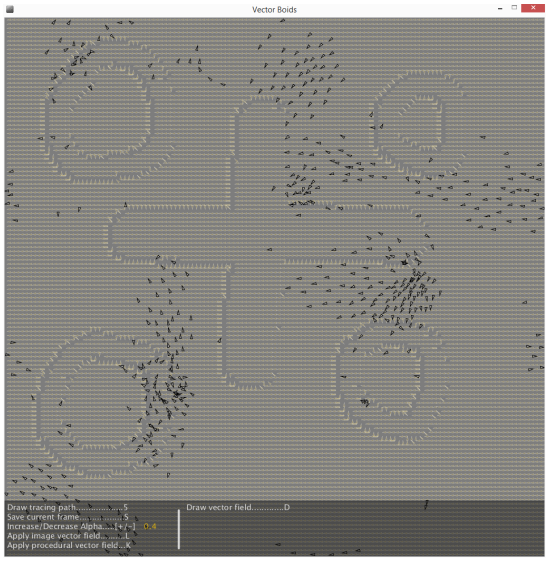


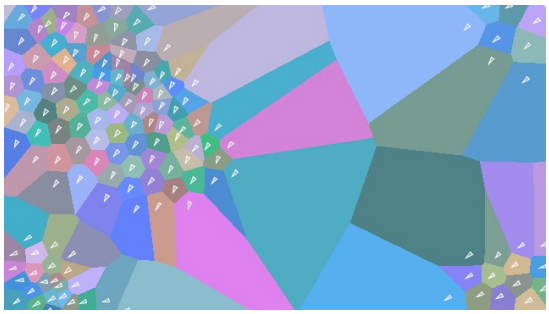
Figure 5.7. A screen shot of the vector framework.

(a) Procedural vector field

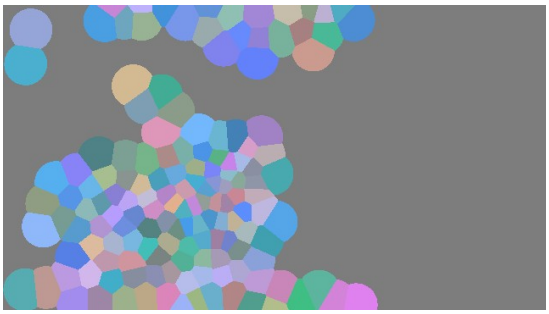


(b) Resulting simulation

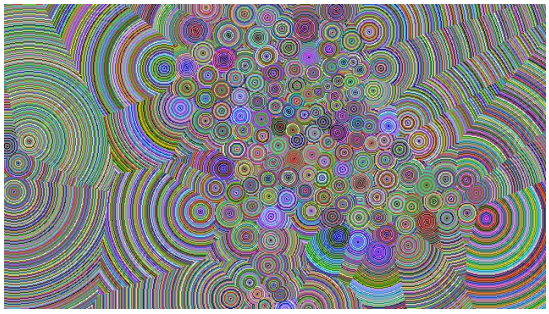
Figure 5.8. User interface to create user defined vector fields.



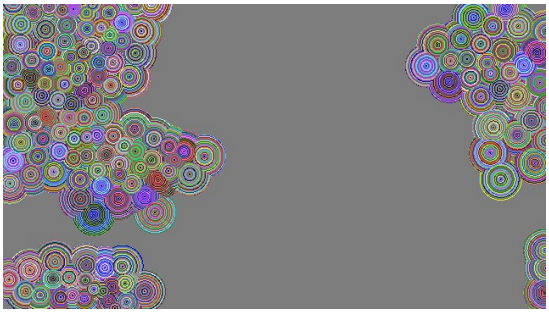
(a) Voronoi



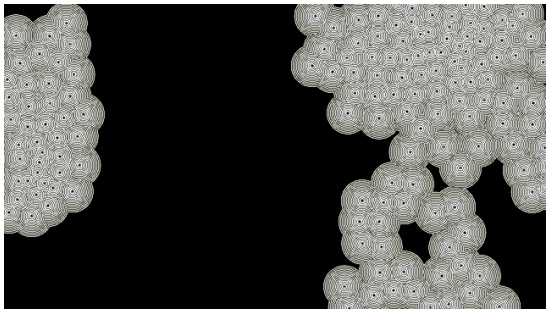
(b) Voronoi



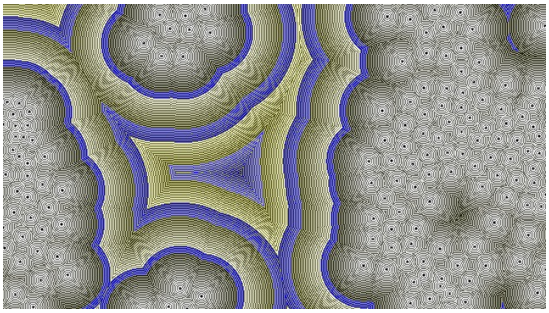
(c) Color manipulation



(d) Color manipulation

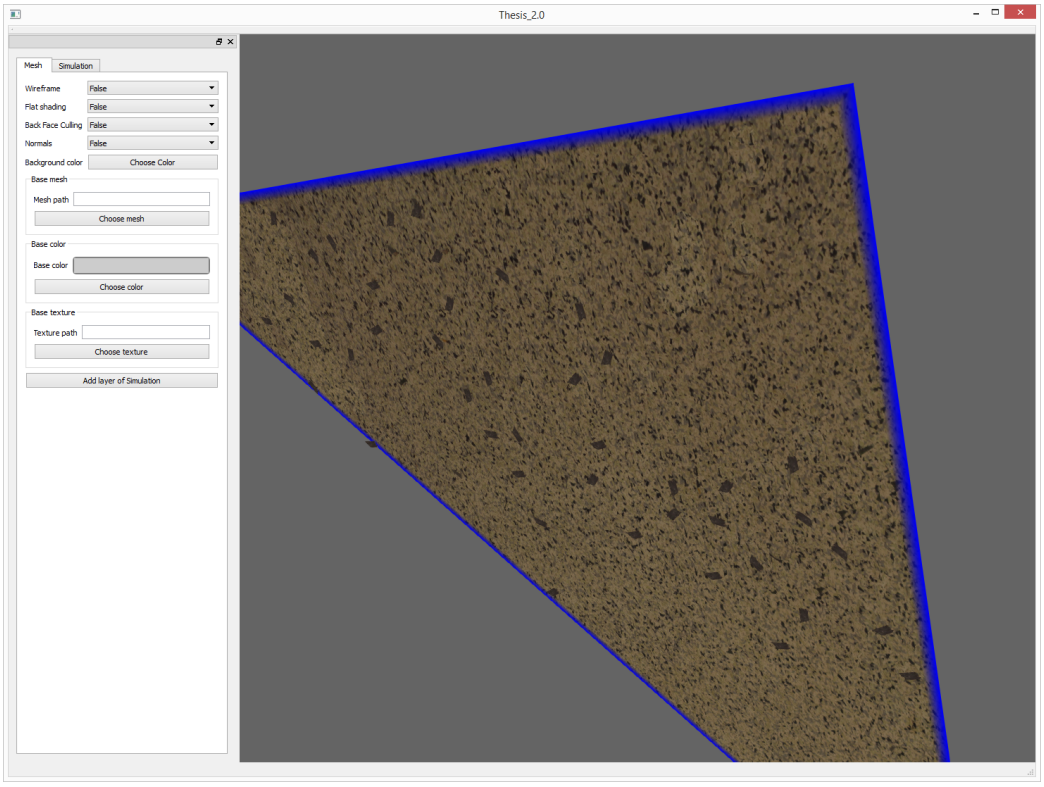


(e) Clustering

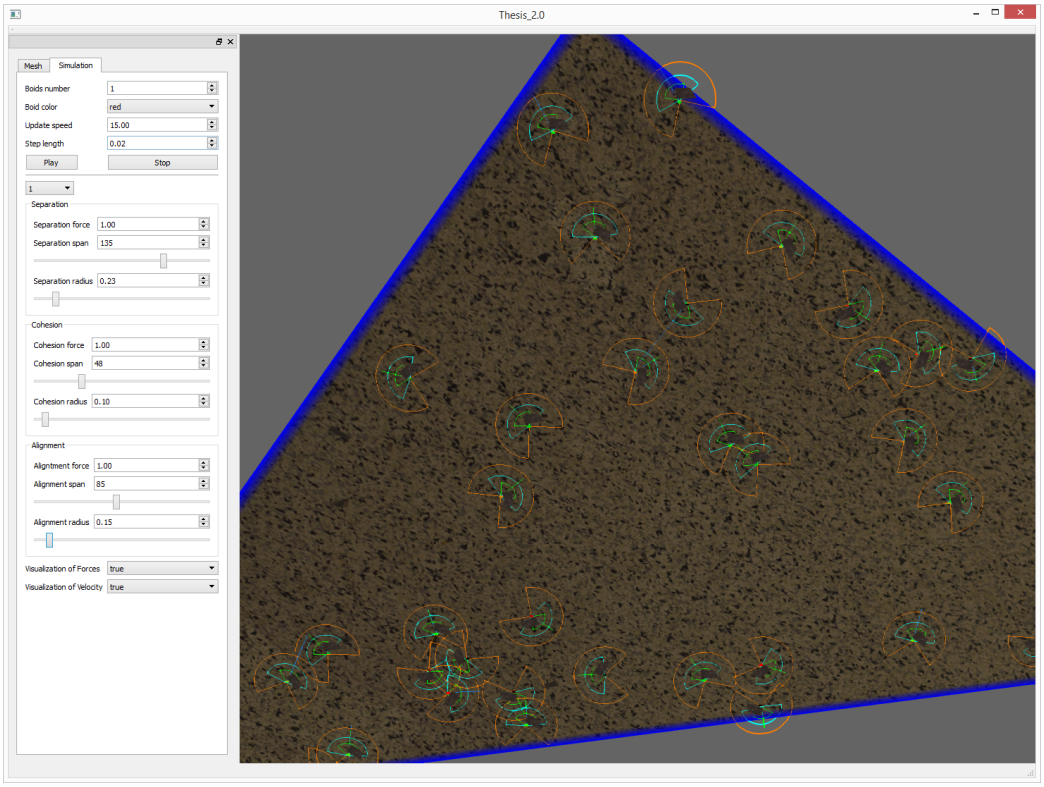


(f) Clustering

Figure 5.9. Results of applying different techniques to boids.



(a) Dark spots are attached features driven by boids.



(b) Boids with visualization of forces turned on.

Figure 5.10. Boids as decals.

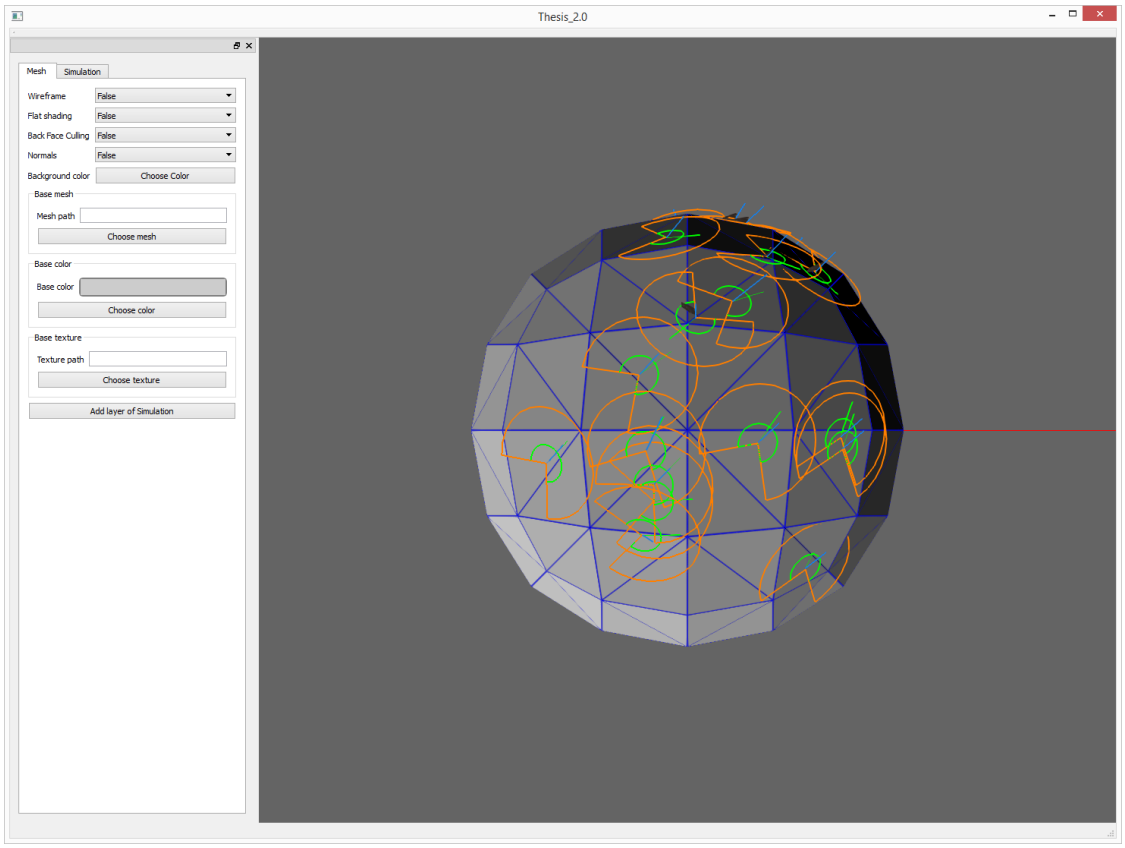


Figure 5.11. Boids are mapped to a surface of a 3D model.

CHAPTER 6. CONCLUSION

6.1 Summary

This research study presented a new approach to texture generation. The approach is based on the behavior described in (Reynolds, 1987). The classical boids' behavior was extended to enhance texture generation. The method of mapping the boids to a 3D surface is developed and allows for boids to interact as they would without constraint. The study defined a class of textures that can be broken down into feature layers, which later can be passed to boids and synthesized. The framework is also extended by combining boids with a vector field.

The study of emerging behavior proved to be filled with new details uncovered at every step. Things that were initially discarded as not important were sometimes reconsidered as significant, deserving a second look. One particular thing that makes research in emergent behavior stand out is that you almost never know how it will turn out in the end. There are endless ways how that the (Reynolds, 1987) behavior as well as the methods presented in this study can be extended. According to the authors, some of the more apparent, are described in the future work section below.

6.2 Challenges

The greatest challenge in this work was caused by numerical errors. Most of the numerical errors were coming from floating point computations. The framework was not initially designed to handle floating point errors appropriately. The accumulation of the errors into linked chains of small errors was causing strange

behavior during the simulation, which was hard to debug. By foreseeing this and building the framework with this in mind could have saved of a lot time and effort.

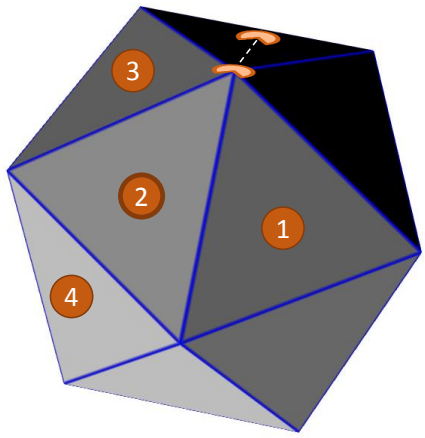


Figure 6.1. Shows current boid's triangle and its neighbors with a boid continuing into a triangle that is not its neighbor.

Another challenge was the edge cases of the simulation. The difficultly of appropriately responding to the edge cases was sometimes closely connected to the challenges with the floating point computation covered above. An example of a difficult edge case would be when a boid would be positioned right on top of a vertex and tries to query its neighbors to cross the edge. This sometimes would cause the boid to transfer into a triangle in front of it, as it should, but that triangle is not one of the neighbors causing an error. The edge case is illustrated in Figure 6.1.

6.3 Possible extensions and future work

There are many directions where the work presented here can be extended. The distance calculation between the boids can be improved by calculating geodesic distance instead of Euclidean distance. The geodesic distance extension also improve the quality of textures by projecting the decals on the surface more accurately. Another way the framework can be improved is by using volume

subdivision algorithms to further reduce the algorithmic complexity. The introduction of boids into generation of normal maps as well as volume textures would also be very interesting to research.

LIST OF REFERENCES

LIST OF REFERENCES

Ashikhmin, M. (2001). Synthesizing natural textures. In *Proceedings of the 2001 symposium on interactive 3d graphics* (pp. 217–226). New York, NY, USA: ACM.

Beneš, C. H. B. (2006). Autonomous boids. *Computer Animation and Virtual Worlds*, *17*(3-4), 199–206.

Bourke, P. (2006). Constrained diffusion-limited aggregation in 3 dimensions. *Computers & Graphics*, *30*(4), 646 - 649.

de Groot, E., Wyvill, B., Barthe, L., Nasri, A., & Lalonde, P. (2014). Implicit decals: Interactive editing of repetitive patterns on surfaces. *Computer Graphics Forum*, *33*(1), 141–151.

Dischler, J.-M., Maritaud, K., Lévy, B., & Ghazanfarpour, D. (2002). Texture particles. In *Computer graphics forum* (Vol. 21, pp. 401–410).

Fleischer, K. W., Laidlaw, D. H., Currin, B. L., & Barr, A. H. (1995). Cellular texture generation. In *Proceedings of the 22nd annual conference on computer graphics and interactive techniques* (pp. 239–248). New York, NY, USA: ACM.

Hillis, W. D. (1988). Intelligence as an emergent behavior; or, the songs of eden. *Daedalus*, *117*(1), pp. 175-189.

Kopf, J., Fu, C.-W., Cohen-Or, D., Deussen, O., Lischinski, D., & Wong, T.-T. (2007, July). Solid texture synthesis from 2d exemplars. *ACM Trans. Graph.*, *26*(3).

Lefebvre, S., Hornus, S., & Neyret, F. (2005, April). Texture sprites: Texture elements splatted on surfaces. In *Acm-siggraph symposium on interactive 3d graphics (i3d).* ACM Press.

Ma, C., Wei, L.-Y., & Tong, X. (2011). Discrete element textures. In *Acm siggraph 2011 papers* (pp. 62:1–62:10). New York, NY, USA: ACM.

Reeves, W. T. (1983, April). Particle systems—a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, *2*(2), 91–108.

Reeves, W. T., & Blau, R. (1985, July). Approximate and probabilistic algorithms for shading and rendering structured particle systems. *SIGGRAPH Comput. Graph.*, *19*(3), 313–322.

Reynolds, C. W. (1987, August). Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph.*, *21*(4), 25–34.

Reynolds, C. W. (1993). An evolved, vision-based behavioral model of coordinated group motion. In *Proc. 2nd international conf. on simulation of adaptive behavior* (pp. 384–392). MIT Press.

Reynolds, C. W. (1994). Competition, coevolution and the game of tag. In *Proceedings of the fourth international workshop on the synthesis and simulation of living systems* (pp. 59–69).

Risser, E., Han, C., Dahyot, R., & Grinspun, E. (2010, July). Synthesizing structured image hybrids. *ACM Trans. Graph.*, *29*(4), 85:1–85:6.

Turk, G. (1991, July). Generating textures on arbitrary surfaces using reaction-diffusion. *SIGGRAPH Comput. Graph.*, *25*(4), 289–298.

Wei, L.-Y., Han, J., Zhou, K., Bao, H., Guo, B., & Shum, H.-Y. (2008, August). Inverse texture synthesis. *ACM Trans. Graph.*, *27*(3), 52:1–52:9.

Witkin, A., & Kass, M. (1991, July). Reaction-diffusion textures. *SIGGRAPH Comput. Graph.*, *25*(4), 299–308.

Witten, T. A., & Sander, L. M. (1983, May). Diffusion-limited aggregation. *Phys. Rev. B*, *27*, 5686–5697.